Final Master Thesis

Màster Universitari en Enginyeria Industrial

Combining Stereo Vision and Deep Learning techniques for object detection in the 3D world

April, 2020

ETSEIB

Author: Gimeno I Jovés, Andrea

Director: Sanfeliu Cortés, Alberto

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria Industrial de Barcelona

Institut de Robòtica i Informàtica Industrial

Summary

The objective of this project is to develop a deep learning algorithm so that, together with the use of a stereo camera, it is capable of detecting a person and locating them in the 3D world.

The person's location in the x-y plane is obtained from the object detector model, which consists of a convolutional neural network, specifically the U-Net, that outputs heat maps.

On the other hand, the person's location in terms of depth (z) is obtained from the depth map given by the ZED stereo camera.

The document begins by presenting the techniques used today for object detection (using heat maps). This is followed by an explanation of the key theory behind neural networks; from the simplest neural networks to the convolutional neural networks. To finish with the theoretical part of the project, the hardware and software equipment used is presented.

To develop and implement the deep learning algorithm, the first thing that is done is the dataset creation. In order to do that, different images have been selected and prepared to enter the network and train the model (using PyTorch) adapted to the needs of this task. Eight different combination of parameters have been used and eight different models have been obtained.

Previously, the metric that will be used to evaluate and compare the different models obtained and choose the one that best suits this application, is defined.

Once the final model is chosen, it is stored in the Jetson AGX Xavier and tested using ZED camera images. In this case, the model is verified to being accurate detecting people and the cases where the algorithm fails are identified.

The next step of this project consists of applying stereo vision techniques to extract the distance at which the detected person is.

A ROS node is created to communicate the ZED camera with the deep learning algorithm. Once the node is ready, it is executed to test the whole program in real time. The ZED color images are passed through the network to detect the person (x, y), and from the ZED depth map, the distance (z) is obtained.

From the results obtained, both for the person detection and for the distance extraction, the existing errors in the designed algorithm are identified, and improvements are made by applying filters and code modifications.

Thanks to the improvements applied to the results, a sufficient precise algorithm is obtained, capable of detecting a person within a distance range in real time.


ETSEIB

Contents

1. Introduction

1.1. Motivation

Artificial Intelligence (AI) has gained popularity in recent years thanks to the increase in computational speed and the amount of useful data available. The most popular and useful subset of AI is the machine learning; a concept that has been feasible only when sufficient amount of data has been available to train machines.

And here comes Deep Learning, a subset of machine learning that has become increasingly popular and important to companies and individual researchers enabling them to overcome challenges that were impossible some decades ago, such as voice recognition or facial recognition.

And we can see how deep learning appears more and more in daily life, as with these two applications that can be found on our mobile devices. Over time, more amazing applications are going to appear, because deep learning is growing every day with high speed. For example, deep learning is increasingly being used for service robots, and it is in this area where the IRI's "Mobile Robotics & Intelligent Systems" department works.

The Institut de Robòtica i Informàtica Industrial, is a Joint Research Center of the Spanish Council for Scientific Research (CSIC) and the Technical University of Catalonia (UPC), conducting basic and applied research in human-centered robotics and automatic control. The institute participate in a large number of international collaborations and cooperate with the community in industrial technological projects.

So, being able to do the project at IRI will give me the opportunity to get in touch with real projects, in addition to entering the world of research.

This IRI's department is working on projects to implement in service robots. In these cases, human-robot interaction is very important and they use different techniques to detect people and obstacles, track and approach them.

Previously, a laser-based system was used to detect people. Nowadays, deep learning techniques are being introduced to detect people and then be able to approach them knowing their location. Until now, these two systems were kept separately; one that detects the person and the other that calculates depth (Figure 1-1).

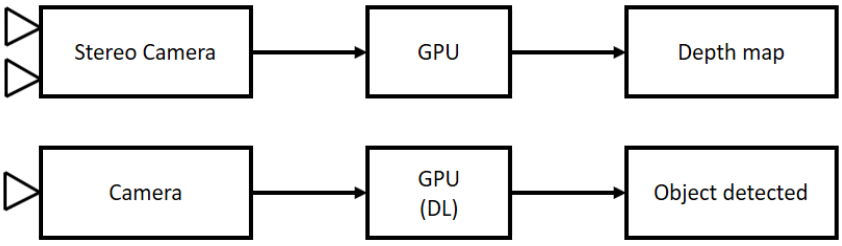


Figure 1-1: Above the Stereo Vision system, and below, the Deep Learning process.

This project seeks to find a solution that combines these two systems (Figure 1-2), that can work in real time reducing costs and with high precision. In this way, by combining deep learning algorithms and stereo vision techniques, the person can be detected and located using only one camera and a GPU.

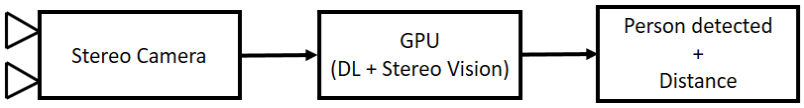


Figure 1-2: Combining Deep Learning and Stereo Vision.

Today there are many open source neural networks and deep learning models. The neural network that we are going to use in this project to detect people, is a network built by an IRI member. This network was created to detect a very specific object.

In this project with deep learning work, we are going to adapt this network to be able to detect people. Therefore, this model will be modified for the new implementation. In addition, adding the stereo vision, it will be useful for different service robot applications, such as approaching a person to give him a package, to take something together and for many other things.

Therefore, the topic of the thesis is chosen to see how deep learning techniques work and the usefulness they have in this type of application for service robots.

In addition, nowadays, deep learning is considered the future of AI and represents the state of the art in the Computer Vision field. Many companies around the world are starting to implement it; having knowledge about deep learning will be useful for my future.

1.2. Objectives

The main objective of this project is to enable a robot to detect a person in an image using deep learning techniques, and measure the distance to that person, using a stereo camera, to know how far it is to approach. To do this, some sub-objectives must be met.

- Develop a real-time object detector using deep learning techniques.
- Run this system on an NVDIA Jetson AGX Xavier so that it can be used on a robot.

- Use the Stereolabs ZED stereo camera to measure the distance.
- Create a ROS node to connect the camera and the object detector to get the distance of the desired point in the 3D world.

1.3. Prerequisites

To do this project, it was necessary to do some tutorials and courses:

- **Coursera Deep Learning course**: theory and practical exercises to know more about Deep Learning. In this course there are the instructions to create a neural network, train and test the network, and some techniques to improve it.
- **PyTorch tutorials**: deep learning framework to program the algorithm.
- **ROS tutorials**: necessary to connect the camera and the object detector.

1.4. Scope of the project

This project considers the implementation of a neural network to detect people in images using PyTorch. And also, the implementation of a ROS node capable of communicating with the ZED camera to obtain the distance of the person.

The design of the algorithm of approaching to the person will not be developed in this project.

Also, the method to select different people or a specific person in the image is not solved in this project. In this project, the deep learning algorithm detects only one person in the image and, if there is more than one, it continues to detect only one; the one that can detect more accurately.

1.5. Thesis layout

Section 2 shows a summary of state of the art techniques for object detection using convolutional neural networks. Section 3 shows the project plan using Gantt diagrams. Section 4 presents some theory needed to understand how convolutional neural networks work. Section 5 introduces the hardware and software required for the project. Section 6 explains the actual tasks that have been performed during the project, including the deep learning techniques and stereo vision. Section 7 presents the results obtained, the errors identified and the improvements implemented. Section 8 presents the proposed project budget. Section 9 discusses the environmental impact of the project. Section 10 summarises the final evaluation of the object detector model.

2. State of the art

Nowadays, deep learning represents the state of the art in the Computer Vision field, with convolutional neural networks (CNN) being the most developed and used algorithms.

Object detection is a core task in Computer Vision, as it is applied in many real-world applications such as surveillance, autonomous driving and robotics. The objective of object detection is to detect the object in the image and localize it. Especially in this project, the algorithm locates a person by giving its coordinates in the 3D world.

Every year a lot of new solutions appear within the Deep Learning community for object detection. Below is a list of some deep learning algorithms and CNN architectures for performing the object detection task, especially, for face detection using heat maps:

- **CenterNet [1]:**

CenterNet models an object as a single point; the center point of its bounding box (Figure 2-1). Then, other properties such as object size, dimensions, 3D location, orientation, and pose are directly obtained from the image features at the center location. Therefore, it becomes a keypoint estimation problem.

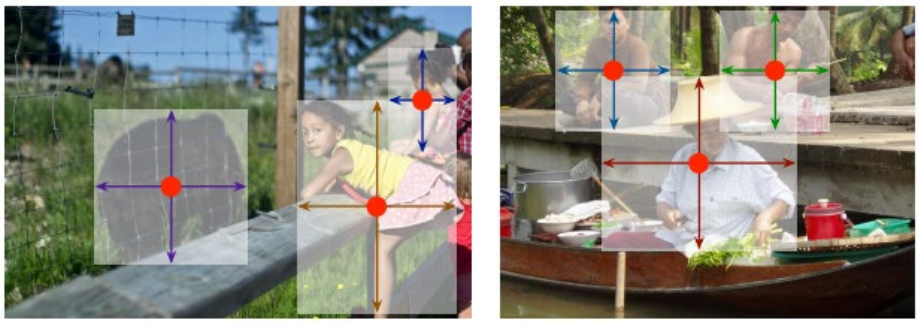


Figure 2-1: Object modelled as the center point of its bounding box [1].

From the input image (WxHx3), a keypoint heat map is produced: $\hat{Y} \in [0,1]^{\frac{w}{R} x \frac{H}{R} x C}$, where R is the output stride and C is the number of keypoint types. This heat map can contain different keypoint types; there are 17 keypoint types for human joints and 80 keypoint types for object categories. A prediction $\hat{Y}_{x,y,c} = 1$ corresponds to a detected keypoint, while $\hat{Y}_{x,y,c} = 0$ is background. Then, a Gaussian kernel is applied to all ground truth keypoints.

To predict the heat map, in this paper [1] they use several different CNNs:

Stacked hourglass network: This network downsamples the input by 4x, followed by two sequential hourglass modules. Each hourglass module is a symmetric 5-layer down- and up-convolutional network with skip connections. This network is quite large, but generally yields the best keypoint estimation performance.

Deep layer aggregation (DLA): An image classification network with hierarchical skip connections. The fully convolutional upsampling version of DLA with some modifications is used in this paper. In this case, the accuracy is lower than when using the previous network (Table 2-1).

Backbone	AP	FPS
Hourglass-104	64.0	6.6
DLA-34	58.9	23

Table 2-1: Average Precision and FPS for both networks [1].

Both previous networks generate heat maps from the input images. Peaks in the output heat map correspond to object centers. Image features at each peak predict the height and weight of the objects bounding boxes. Therefore, from this keypoint estimator, the 3 outputs shown at the top of Figure 2-2 can be estimated.

In addition, this method also provides 3D object detection and human pose estimation, by predicting additional outputs at each center point (Figure 2-2). To do so, the algorithm requires additional attributes per center point, such as the depth value, 3D location, its dimension, orientation and the human joints locations. By introducing some modifications to the previous CNN used and implementing other loss functions, these outputs can be estimated.

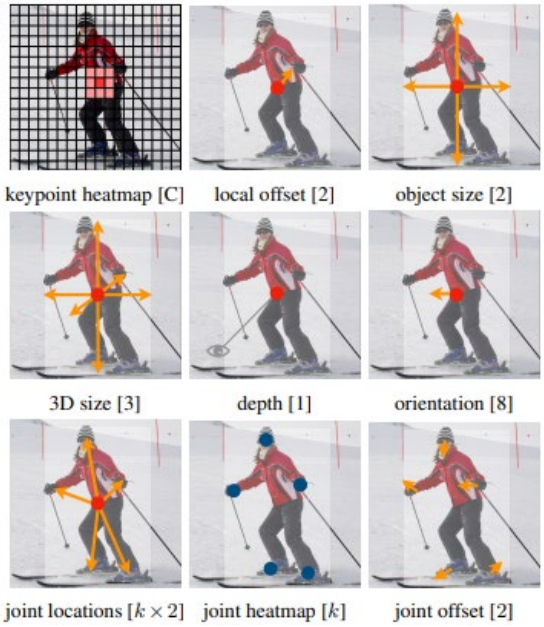


Figure 2-2: NN outputs: top for object detection, middle for 3D object detection, bottom: for pose estimation [1]

Object detectors generally identify objects as axis-aligned boxes in an image. The most successful object detectors, like YOLO or Faster R-CNN, enumerate a nearly exhaustive list of potential object localizations and classify each one. For each bounding box, the classifier

determines whether the image content is a specific object or background. In the end, this seems to be wasteful, inefficient, and requires additional post-processing.

Comparing CenterNet with these bounding box based detectors on the MS COCO dataset, it turns out that CenterNet is much simpler, faster and more accurate.

- CenterNet with Hourglass-104 achieves the best accuracy at a relatively good speed, with a 42,2% AP in 7,9 FPS.
- CenterNet with DLA-34 gives the best speed/accuracy trade-off. It runs at 52 FPS with 37,4% AP. This is more than twice as fast as YOLOv3 and 4,4% AP more accurate.

- **Convolutional Neural Network with heat maps [2]:**

In this paper [2], a heat map approach is applied to track human faces. The heat map extracted from the CNN is used for face/non-face classification problem. In this case, the sliding window methods (as YOLO) are also avoided for reducing time-consuming.

The CNN architecture is shown in Figure 2-3. This network is built in order to extract information that is meaningful in locating an object.

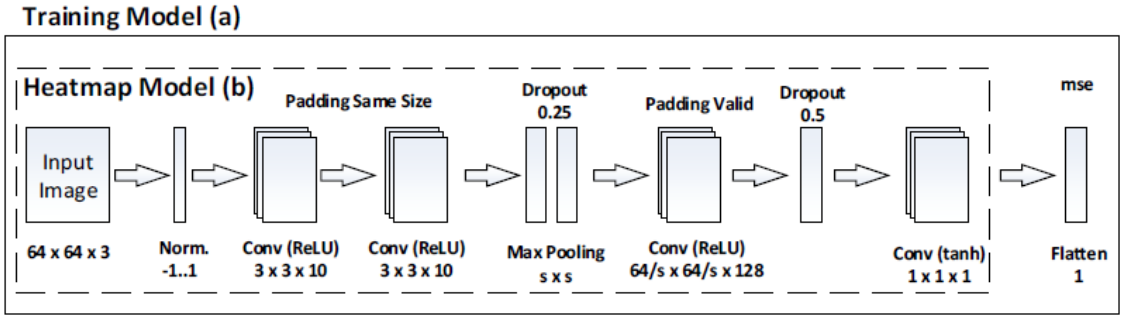


Figure 2-3: CNN for binary classification face/non-faces [2].

In this paper [2] a simple CNN for binary classification problem is built, able to predict the semantics and the face location.

The network input is an image of size 64x64x3. Then there are two convolutional layers 3x3 with 10 features. Next, there is the max pooling layer with size *s* x *s* that is used to reduce the image size, as well as to highlight important features. There are also two dropout layers to prevent overfitting. The last two layers play the role as fully connected layers. The first one has a size of 64/s with no padding, and 128 features. These 128 features are connected with the last convolutional layer, which has a size of 1x1x1. The third dimension is 1 because represents the binary classification operation; 1 if it is a face, and -1 if not. These two layers are common when generating heat maps for object detection.

They proposed a method to build heat maps from many CNNs with different pool-sizes. In the paper, they experiment with different pool-sizes of the max pooling layer to improve the model accuracy. By increasing the pool-size, better determination in special details was achieved, but the accuracy in determining face/non-face regions on the heat map decreased.

Therefore, the proposed final heat map is based on a feedback rating from the CNNs according to the respective weights. From there, using a threshold value, a mask is built to quickly identify objects in the target tracking window (Figure 2-4).

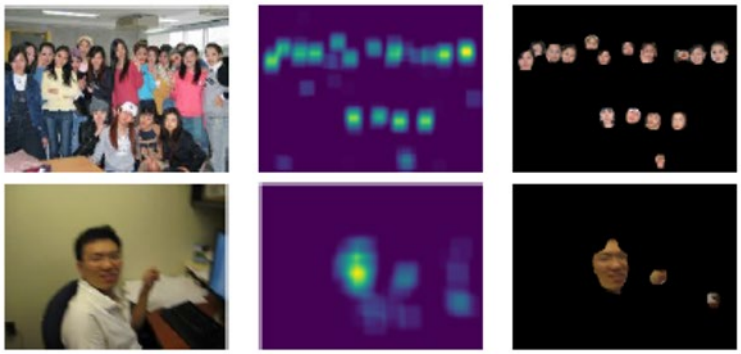


Figure 2-4: On the left, images containing faces, in the middle, the heat-maps obtained from CNNs feedback, and on the right, the mask image identifying human face based on the specified threshold [2].

As seen in Figure 2-4, this algorithm gives good results in determining the face mask image. However, there are some situations where the heat map determines wrong human face regions.

- **U-Net for face detection with segmentation maps [3]:**

A deep convolutional neural network is trained to encode the position and scale of the objects into a segmentation map. Then, from this map, the algorithm also decodes the bounding boxes of the objects. In this case, this algorithm is used to build a face detector (Figure 2-5).

Figure 2-5: On the left, the input image and on the right, the corresponding segmentation map.

The idea is to place a diagonal line segment across each face in the input image. The location of the segment centroid corresponds to the location of the face. The segment length specifies the face size. From this encoding scheme, it is easy to decode the objects bounding boxes in run-time.

This method based on semantic segmentation is also less complex than typical object detectors like YOLO and Faster R-CNN.

Semantic segmentation is the process of labeling each pixel in the image with a class label from a predefined set of classes. Semantic segmentation is useful when a detailed understanding of the image is required. For example, it is used for medical image processing, i.e. to locate diseases.

To solve the segmentation task, a network that outputs a prediction of the same size as the input image, is needed. These types of networks are desired when using segmentation or, for example, for the depth prediction task. A particularly effective architecture is the U-Net design (Ronneberger et al, 2015).

U-Net was first designed especially for medical image segmentation. It showed such good results that is used in many other fields after. This NN architecture is used when you need to convert a feature map into a vector and then reconstruct an image from this vector.

This architecture follows the usual encoder-decoder scheme but uses feature maps computed in the earlier downscaling steps during each upsampling process (this reusage of cached features is shown in Figure 2-6). As while converting an image into a vector, the feature maps has been already learned, these same maps are used to convert the vector again to image. Therefore, the same feature maps that are used for contraction are used to expand the vector to a segmented image. And this would preserve the structural integrity of the image, reducing distortion.

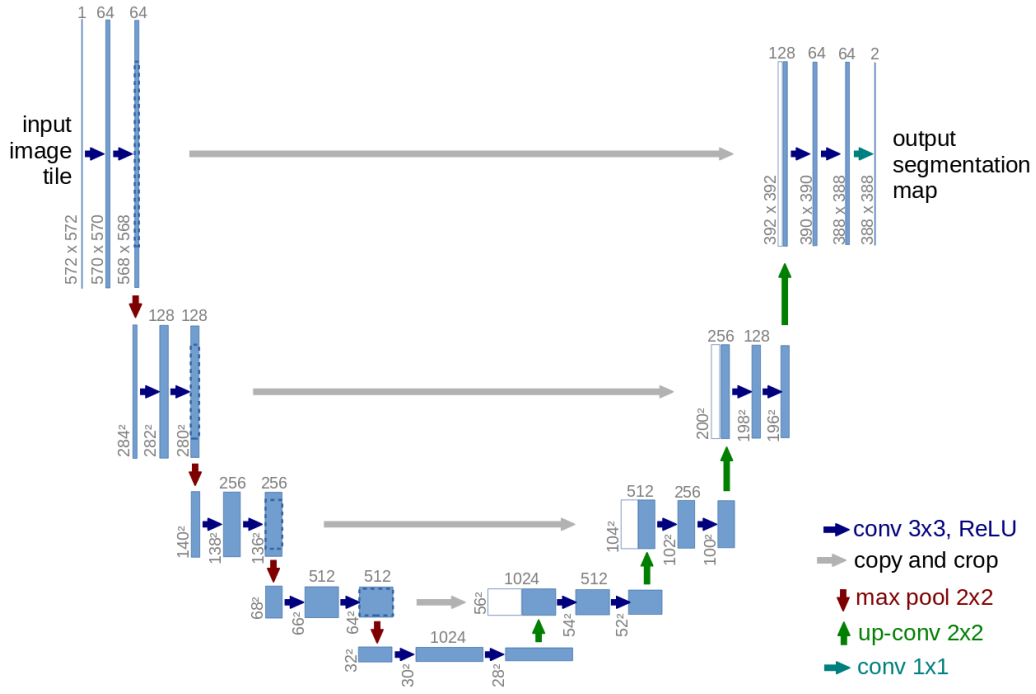


Figure 2-6: U-Net architecture [6]

As can be seen in Figure 2-6, the architecture looks like a “U”. This architecture consists of two sections: the contraction (left side) and the expansion section (right side).

The contraction section follows the typical architecture of a CNN and consists of the repeated application of contraction blocks. Each block takes an input and applies two 3x3 convolutional blocks (a convolutional layer (with no padding), followed by a ReLU and a normalization layer) and a 2x2 max pooling layer (with stride 2 for downsampling). The number of filters (output channels) after each block is doubled, so that architecture can learn the complex structures effectively [6].

Every step in the expansive section consists of an upsampling of the feature map followed by a 2x2 convolution (“up-convolution”) that halves the number of feature channels, a concatenation with the corresponding cropped feature map from the contracting section, and two 3x3 convolutional blocks (a convolutional layer followed by a ReLU and a normalization layer). The cropping is necessary due to the loss of border pixels in every convolution.

At the final layer, a 1x1 convolution is used to map each 64 features to the desired number of classes.

In total, the original U-Net has 23 convolutional layers.

In this case [3], they use a modification of this architecture to make the model smaller in order to achieve real-time detection rates.

The predicted output map has all its pixels classified into one of the two classes: 1 belongs to a line segment and 0 otherwise. This output segmentation map with its values in the range [0,1], is obtained by applying a sigmoid function to each element of the previous feature map.

The training dataset must contain images of faces with its bounding boxes and, consequently, ground truth segmentation masks. They train its model using RMSProp with a learning rate set to 0,0001, using mini-batches and 500 epochs.

Finally, this model is capable of processing frames in real time.

- **U-Net for face detection with heat maps [4]:**

This research, carried out by IRI (Institut de Robòtica I Informàtica) staff [4], presents a methodology for detecting the crawler used in the AEROARMS project. The crawler is required for outdoor industrial inspection and maintenance. An aerial robot picks up and releases the crawler in areas or structures that require some inspection or maintenance but are inaccessible, very dangerous or costly to be accessed from ground.

For this project, deep learning techniques have been applied to detect the crawler centre point. To do it, different networks have been tested. First, two CNNs were used to locate the crawler

and to output the crawler probability of being in the image. However, the U-Net architecture has been used locating the crawler in real-time, with especially great results.

In this case, the U-Net inputs an RGB image and outputs a heat map; a probability distribution map. The peak in this map represents the point of the crawler where the aerial robot picks it up.

For that project, a small U-Net with 14 convolutional layers was also built. The code can be found on GitHub [5]. This tiny U-Net has the following architecture (Annex 12.1):

First, the input passes through two 7x7 convolutional blocks (a convolutional layer, followed by a normalization layer and a ReLU) and then trough a 2x2 average pool layer.

Then, the input passes through five contraction blocks. Each block passes the input though a single 3x3 convolutional block (a convolutional layer, followed by a ReLU and a normalization layer) and then, though a 2x2 average pool layer. The number of filters (output channels) after each block, is doubled. In this case, there is only one convolutional block for each contraction block, instead of two.

Then, the feature map passes through five expansion blocks. Each block takes the input feature map and upscales it by a factor of 2. Then, it concatenates this feature map with the feature map from the corresponding contraction block. And then, it passes this new input feature map through a single 3x3 convolutional block (a convolutional layer, followed by a ReLU and a normalization layer). In this case, there is only one convolutional block for each expansion block.

Finally, to obtain the predicted heat map, the feature map passes through a 3x3 convolutional block (a convolutional layer, followed by a ReLU and a normalization layer) followed by a 3x3 convolutional layer. The output size dimensions and the layer's parameters are specified in Annex 12.1.

In this case, there are only 14 convolutional layers. Conversely, in the original U-Net there are 23 convolutional layers. As can be seen in the annex, these layers are the ones that have trainable parameters. Therefore, this small U-Net has less trainable parameters; around 11 million parameters (Annex 12.1), while the original U-Net has around 40,5 million parameters.

3. Project plan

This project starts in July 2019 and has been planned to be finished in April 2020 (Figure 3-1).

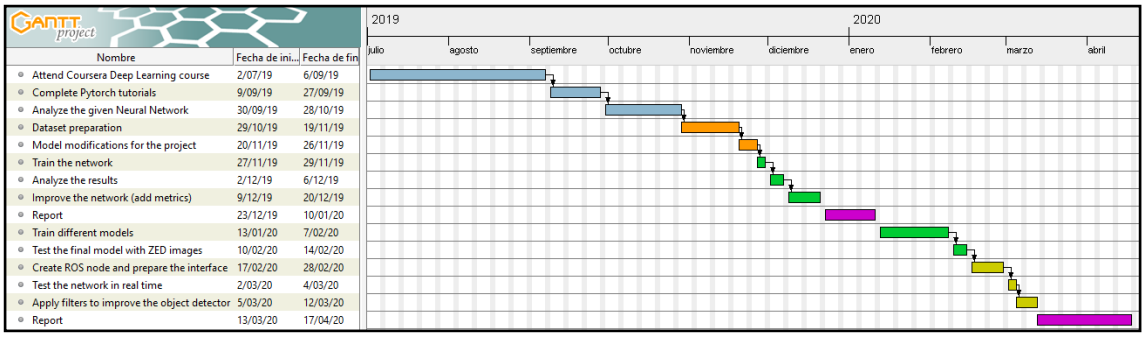
The blue tasks are the formation and documentation tasks. Among them are the courses and tutorials mentioned above and the analysis of the given neural network; it is necessary to understand and learn how this network and the object detector model work.

The orange tasks are those related to preparing the dataset and adapting the object detector model for this project.

The green tasks are those related to the implementation of the object detector; including training the network, analyzing the results, improving the model and adding metrics to evaluate the model, and testing the model with new images.

The yellow tasks are those related to stereo vision techniques. These tasks include creating the ROS node, testing the algorithm in real time and applying filters to improve the algorithm.

The purple task represents the report development (elaborate the draft, produce the draft).



Nombre	Fecha de ini...	Fecha de fin
Attend Coursera Deep Learning course	2/07/19	6/09/19
Complete Pytorch tutorials	9/09/19	27/09/19
Analyze the given Neural Network	30/09/19	28/10/19
Dataset preparation	29/10/19	19/11/19
Model modifications for the project	20/11/19	26/11/19
Train the network	27/11/19	29/11/19
Analyze the results	2/12/19	6/12/19
Improve the network (add metrics)	9/12/19	20/12/19
Report	23/12/19	10/01/20
Train different models	13/01/20	7/02/20
Test the final model with ZED images	10/02/20	14/02/20
Create ROS node and prepare the interface	17/02/20	28/02/20
Test the network in real time	2/03/20	4/03/20
Apply filters to improve the object detector	5/03/20	12/03/20
Report	13/03/20	17/04/20

Figure 3-1: Project plan

4. Key theoretical concepts

4.1. Neural network basics

4.1.1. Introduction

Artificial intelligence (AI) is the computer science branch that is focused on building smart machines capable of performing tasks that generally require human intelligence. In other words, it tries to simulate human intelligence in machines. AI is an interdisciplinary science with multiple approaches, which is progressing rapidly, and it is being implemented in many sectors and giving very good results. Advances in machine learning, and deep learning in particular, are the most surprising; bringing about a huge transformation and creating a paradigm shift in almost every sector of the technology industry [7].

Machine learning (ML) is an important branch of AI, and specifically of computer science. The part in ML that is rising rapidly and driving a lot of these changes is deep learning (DL). Deep Learning has already transformed traditional Internet businesses like web search and advertising. But it is also enabling brand new products and businesses, and ways to help people create. It has found a great success in a lot of applications such as natural language processing, image recognition, speech recognition, machine translation, bioinformatics, medical image analysis, autonomous driving and many others.

So nowadays, deep learning is one of the most highly sought after skills in technology worlds. Especially, deep learning algorithms have proven to be really good for Computer Vision field, with these algorithms being the most widely used today.

Deep learning is taking off due to three factors:

- large amount of data available thanks to the society digitalization
- faster computation (GPUs and CPUs)
- innovation in the development and optimization of the algorithms

Neural networks (NNs) are the main architecture used in deep learning. That is why deep learning is said to be based on the way the human brain processes information and learns. This consists of a neural network model composed of several levels of representation, where each level uses the information from the previous level to learn [8]. Deep learning uses neural networks to provide accurate results.

To better understand this project, it is required to understand NNs. In these following sections, some theoretical concepts are explained following the Deep Learning Specialization course from Coursera [9].

To start, the best example to explain what is a neural network, is the supervised deep learning task of image classification (Figure 4-1).

These tasks consist in telling if there is a specific object in an image. Then, the input (x) will be an image and the desired output (y) will be a discrete value (a class); 1 if the object is in the image, and 0 otherwise.

In supervised learning, during the training phase, a set of examples (training images) is submitted as input to the system. Each input is labeled with a desired output, and the system knows the output when the input is submitted. For example, the NN will know if the input image contains the specific object or not. Then, during the training, given the input (x) the NN will try to predict ($\hat{y}$) the probability of being the specific object in the input image ($P(y = 1|x)$). Then, the training is performed by the minimization of a particular cost function which represents the error between the output predicted by the network ($\hat{y}$) and the desired output (y) [8].

The simplest possible neural network is formed by a single neuron that implements a function that maps the input x to the output y. The neuron takes the input image labeled (x, y) and outputs the prediction $\hat{y}$.

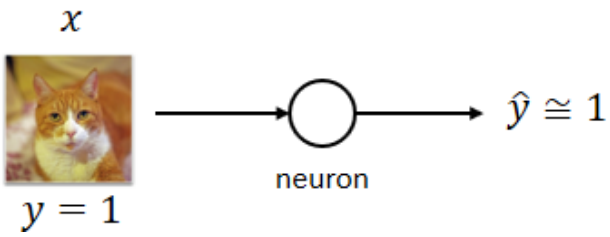


Figure 4-1: Single neuron neural network for a cat classifier

This simple neural network is used in logistic regression.

4.1.2. Logistic Regression

Given an input x, which belongs to a class y, the neural network outputs $\hat{y}$. The goal is to train a classifier that, given an input image represented by its feature vector x, accurately predicts whether it belong to class 0 or 1. For example, considering a cat classifier, the algorithm has to predict whether the input image is a cat image (1) or a non-cat image (0).

An image is stored in the computer as an array of values. Typically, it's a 3-dimensional matrix of pixel values (W, H, C), adding the third dimension for the Red, Green and Blue channels (Figure 4-2). The value in a cell represents the pixel intensity which will be used to create a feature vector of n_x dimension. To create the feature vector x, the pixel intensity values will be unrolled for each channel, and if the image shape is 64x64, the dimension of the input feature vector will be n_x=64x64x3.

Figure 4-2: Representation of an image and its feature vector

A single training example is represented by a pair (x,y) where x ($x \in \mathbb{R}^{n_x}$) is the n_x-dimensional feature vector and y ($y \in \{0,1\}$), the label or class, is either 0 or 1.

However, if the entire training set is comprised by *m* training examples, the training set will be represented by *m* pairs (*x,y*), one for each training example. Finally, to have all of the training examples into a compact structure, the matrix X ($X \in \mathbb{R}^{n_x xm}$) is defined by taking all the inputs (x_i) and staking them in columns. It would be convenience to also define a matrix Y ($Y \in \mathbb{R}^{1xm}$) by staking the labels (y_i) of each training example in columns.

Once the notation has been defined, the logistic regression algorithm can be explained.

Considering a cat classifier (Figure 4-3), given the input feature vector x, which belongs to a class y, the algorithm outputs a prediction $\hat{y}$, which is the estimation of y. More formally, $\hat{y}$ is the probability of the input feature belonging to class 1 (Eq. 4-1).

$$\hat{y} = P(y = 1|x), \text{ where } 0 \leq \hat{y} \leq 1 \qquad \textit{Eq. 4-1}$$

The goal of logistic regression is to minimize the error between its predictions ($\hat{y}$) and training data (*x,y*).

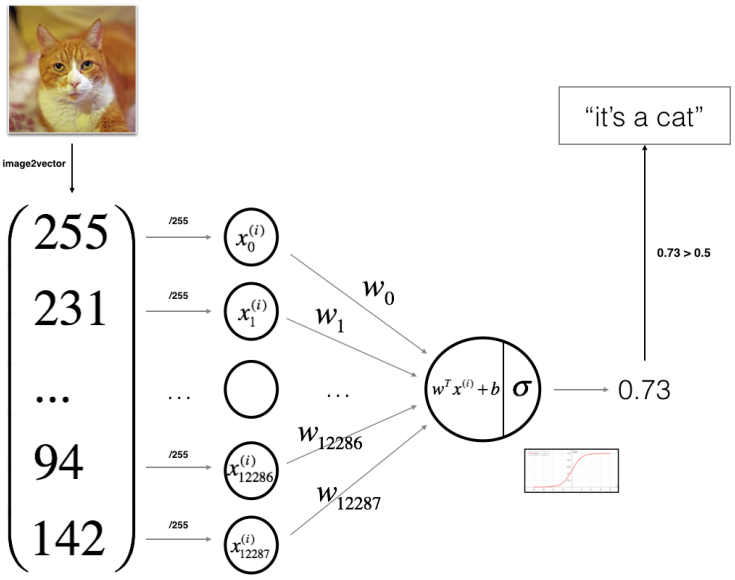


Figure 4-3: Architecture of cat image classifier [10]

The computational elements of NNs are the neurons. Each neuron takes all its inputs and compute its output, first, using a linear function and then, applying a non-linear function to the linear result. For example, for the cat classifier example, the neuron outputs $\hat{y}$, according to Eq. 4-2 and Eq. 4-3.

$$z = w^T x + b \qquad \textit{Eq. 4-2}$$

$$\hat{y} = a = g(z) \qquad \textit{Eq. 4-3}$$

The Eq. 4-2 is the linear function. The parameters of logistic regression appear in this equation and are known as the weights w ($w \in \mathbb{R}^{n_x}$) and the bias b ($b \in \mathbb{R}$). Then, the Eq. 4-3 is used to pass the linear result (which can be greater than 1 or it can even be negative) into a non-linear function. This function is called activation function and its output a is called activation. Different non-linear functions have been historically used. Since we are looking for a probability constraint between 0 and 1, a reasonable activation function to use would be the sigmoid function (Eq. 4-4), which is bounded between [0,1]. The sigmoid function takes the linear result and transforms it into a value between 0 and 1.

$$a = \sigma(z) = \frac{1}{1+e^{-z}} \qquad \textit{Eq. 4-4}$$

The sigmoid function (Figure 4-4) goes smoothly from zero up to one and it crosses de vertical axis at 0,5. If z is a large positive number, the sigmoid function of z will be approximately 1 and, conversely, if z is a large negative number, then the sigmoid function of z goes very close to 0.

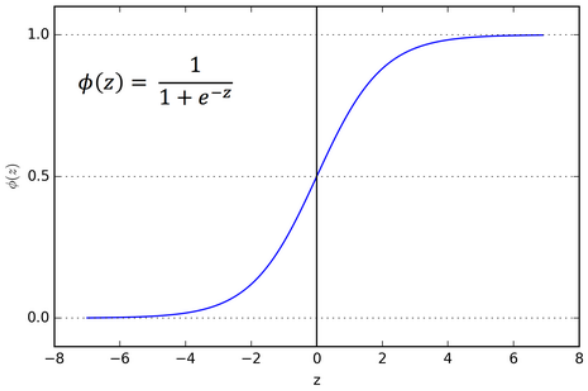


Figure 4-4: Sigmoid function [11]

The objective of applying logistic regression is to learn the parameters w and b so that $\hat{y}$ becomes a good estimate of the chance of y being equal to 1. These parameters are the main tool that enables NNs to provide such accurate outputs. To learn the parameters w and b of the logistic regression model, it is necessary to define a cost function.

4.1.3. Cost function

Given a set of *m* training examples:

$$\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$$

It is desired:

$$\hat{y}^{(i)} \approx y^{(i)} \qquad \textit{Eq. 4-5}$$

Where the superscript $i \in [1, m]$ refers to data associated with the $i\text{-}th$ training example, being *m* the length of the training dataset.

The cost function is used to measure how well the algorithm is doing. As the Eq. 4-5 states, the more similar the predictions $\hat{y}$ and the labels y are, the more accurate the algorithm is.

To define the cost function, first it is necessary to define the loss function. First, let's rewrite the Eq. 4-2 and Eq. 4-3 considering that they can be evaluated in different samples (Eq. 4-6).

$$\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b) \qquad \textit{Eq. 4-6}$$

The loss function L is the function used to measure the discrepancy between the prediction $\hat{y}$ and the desired label y. In other words, it is used to measure how good the prediction $\hat{y}$ is when the true label is y.

One option is to define the loss function to be the one half square error (Eq. 4-7):

$$L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2 \qquad \textit{Eq. 4-7}$$

As stated above, the goal is to find the parameters w and b that make the predictions $\hat{y}$ close to the labels y and, therefore, make the error close to zero. To do so, an optimization algorithm called gradient descent is used. Nevertheless, this loss function is not normally used because, when the algorithm tries to learn the parameters, the optimization problem becomes non-convex (with multiple local optima) and thus, the global optimum may not be found. For that reason, what is actually used in logistic regression is a different loss function that gives a convex optimization problem and becomes much easier to optimize (Eq. 4-8).

$$L(\hat{y}, y) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \qquad \textit{Eq. 4-8}$$

In the learning procedure, this loss function is desired to be as small as possible.

Finally, the loss function computes the error for a single training example while the cost function J is defined to measure how well the algorithm is working with the entire training set.

$$J(w, b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})] \qquad \textit{Eq. 4-9}$$

The cost function is the average of the loss function of the entire training set, where $\hat{y}$ is the

prediction output of the logistic regression algorithm, computed, according to the *Eq. 4-9*, using a particular set of parameters w and b. Consequently, when training the logistic regression model, it is required to find the parameters w and b that minimize the overall cost function using the gradient descent method.

4.1.4. Gradient descent method

In order to achieve good predictions, the gradient descent algorithm is used to train the parameters w and b, by minimizing the overall cost function J (Eq. 4-9).

A simplified representation of this procedure can be seen in Figure 4-5. In this graphic, the horizontal axes represent the spatial parameters w and b. In practice, w can be much higher dimensional, but for the purpose of simplifying the plotting, w and b are illustrated as single real numbers. The cost function $J(w, b)$ is, then, a surface above these horizontal axes, so that the height of the surface represents the value of $J(w, b)$ at a certain point. And the goal is really to find the value of w and b that corresponds to the minimum of the cost function J (represented in Figure 4-5 as a red dot).

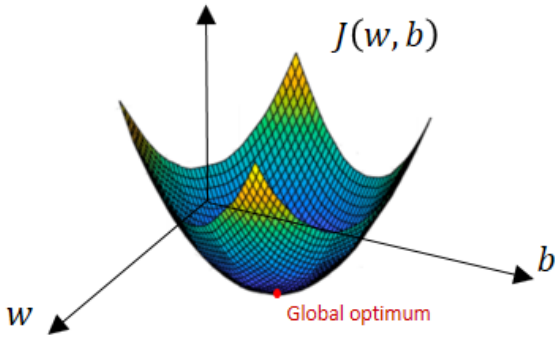


Figure 4-5: Simplified representation of the cost function

Note that the cost function is convex and, as said before, this is one of the huge reasons why this particular cost function is used in logistic regression. So to find the good values for the parameters, w and b are initialized to some initial value and then the gradient descent method is applied. Although almost any initialization method works for logistic regression, the parameters are usually initialized to zero. But because this function is convex, no matter where the initialization is done; the algorithm should get to the same point (or roughly to the same point).

The gradient descent is an iterative algorithm, which starts at that initial point and then takes steps in the steepest downhill direction iteratively until eventually it converges to the global optimum (or, at least, to a very close value). To explain the method, let's illustrate the gradient descent algorithm in Figure 4-6 using a one-dimensional plot (ignoring b for now) and having a function $J(w)$ that you want to minimize.

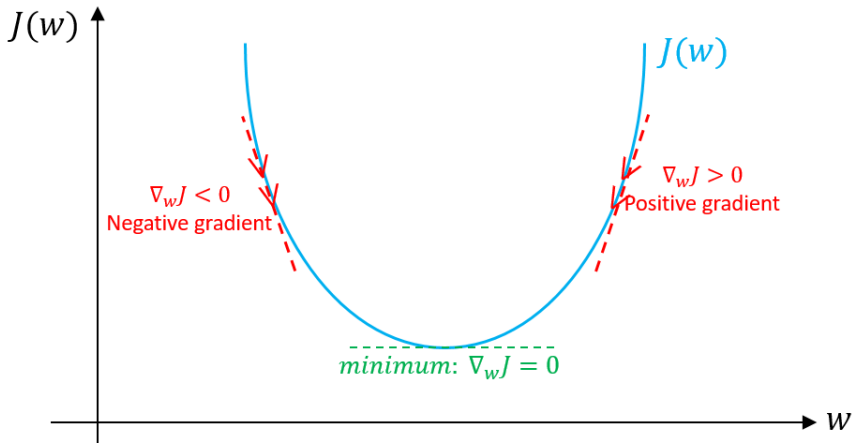


Figure 4-6: Gradient descent method. An illustration of how gradient descent algorithm uses the first derivative of the loss function to follow downhill it's minimum [12].

Once the parameters w and b are initialized, the algorithm repeatedly carries out the following update:

$$w = w - \alpha \frac{dJ(w)}{dw} \qquad \textit{Eq. 4-10}$$

In this equation, the new parameter alpha appears. This variable is called "learning rate" and it is a scalar value, and it controls how big must be the step in each iteration. The learning rate is a tuning parameter that may be very important when configuring the neural network. In the Eq. 4-10, the partial derivative of the cost function $J(w)$ with respect to the parameter w also appears. This derivative term represents the slope of the cost function in a specific point and, when writing the code to implement gradient descent, it is represented by the variable name "dw". It is important to know the slope of the function at the current setting of the parameters to take these steps of steepest descent, in order to go downhill on the cost function.

In Figure 4-6, if the parameter w is initialized with a value that is on the right of the global optimum, the derivative will be positive and the gradient descent will make the algorithm slowly decrease the parameter w. The algorithm will take steps to the left until it converges. On the other hand, if the parameter w is initialized with a value that is on the left of the global optimum, the slope will be negative and the gradient descent update will subtract alpha times a negative number, thus making w bigger with successive iterations. In this case, the algorithm will take steps to the right until it converges on the global optimum. Therefore, no matter where the initialization is done, the gradient descent will move the point towards the global minimum.

Considering that, in logistic regression, the cost function is a function of both w and b parameters, the gradient descent loop that is actually implemented is:

Repeat {

$w = w - \alpha \frac{\partial J(w,b)}{\partial w}$, which is the same as: $w = w - \alpha \cdot dw$

$b = b - \alpha \frac{\partial J(w,b)}{\partial b}$, which is the same as: $b = b - \alpha \cdot db$

}

To sum up, the parameters w and b are initialized (usually to zero) and then, at each step, the gradient of the cost function will be computed to take a step in the steepest downhill direction. With each step, the point would have change to a new point closer to the global minimum and after some iterations, the global minimum would be achieved.

4.1.5. Gradient Descent for Logistic Regression

The NNs computations are organized as follows: a forward propagation step, in which the cost function is computed, followed by a backward propagation step, in which the gradients are computed.

The key equations that are used to implement gradient descent for logistic regression are described below.

For this example, let's consider a NN with one layer with a single neuron (a logistic regression problem) with only two features "x_1" and "x_2" as an input data (Figure 4-7). In order to compute the output of logistic regression, the parameters "w_1" , "w_2" and b need to be initialized.

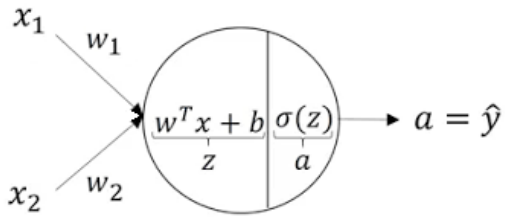


Figure 4-7: Logistic regression model with two features [9]

The neuron in logistic regression computes the output $\hat{y}$ according to Eq. 4-11 and Eq. 4-12.

$$z = w^T x + b = w_1 \cdot x_1 + w_2 \cdot x_2 + b \qquad \textit{Eq. 4-11}$$

$$\hat{y} = a = \sigma(z) \qquad \textit{Eq. 4-12}$$

In the case of logistic regression, the cost function $J(w, b)$ is the function that the algorithm tries to minimize. In the forward propagation step (a left-to-right pass), the value of $J(w, b)$ can be computed (Figure 4-8). Focusing on just one training example for now, the lost function associated with that example is defined as follows:

$$L(\hat{y}, y) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \qquad \textit{Eq. 4-13}$$

And writing the output of logistic regression $\hat{y}$ as "a" according to Eq. 4-13:

$$L(a, y) = -(y \log(a) + (1 - y) \log(1 - a)) \qquad \textit{Eq. 4-14}$$

Then, another step is necessary: the backward propagation step.

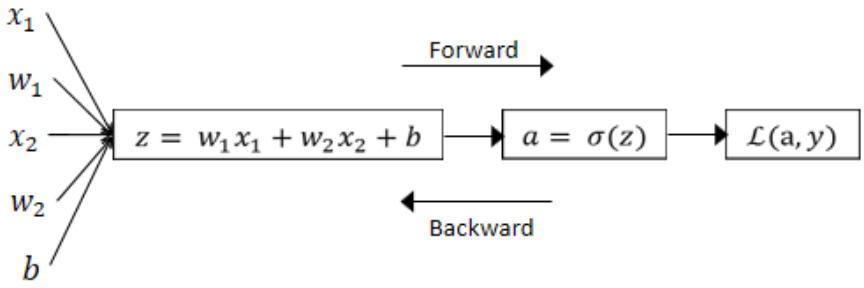


Figure 4-8: Computation graph of logistic regression [9]

Once the loss on the single training example is computed, there is a right-to-left pass (Figure 4-8), the backward propagation step, in order to compute the derivatives with respect to this loss. When going backwards, the derivative of the loss with respect to the variable "a" is computed. This derivative is denoted by "da" and, in this example, it is computed as follows:

$$da = \frac{\partial L(a,y)}{\partial a} = -\frac{y}{a} + \frac{1-y}{1-a} \qquad \textit{Eq. 4-15}$$

Then, this derivative "da" is used to compute the next derivative "dz" (Eq. 4-16).

$$dz = \frac{\partial L(a,y)}{\partial z} = \frac{\partial L(a,y)}{\partial a} \cdot \frac{\partial a}{\partial z} = \left[-\frac{y}{a} + \frac{1-y}{1-a}\right] \cdot [a(1-a)] = a - y \qquad \textit{Eq. 4-16}$$

The final step in backward propagation is to compute how much the parameters w and b should change. Eq. 4-17, Eq. 4-18 and Eq. 4-19, show the derivatives of the loss with respect to all these parameters.

$$dw_1 = \frac{\partial L(a,y)}{\partial w_1} = x_1 \cdot dz \qquad \textit{Eq. 4-17}$$

$$dw_2 = \frac{\partial L(a,y)}{\partial w_2} = x_2 \cdot dz \qquad \textit{Eq. 4-18}$$

$$db = \frac{\partial L(a,y)}{\partial b} = dz \qquad \textit{Eq. 4-19}$$

Finally, to implement gradient descent, the previous derivatives are used to update the parameters as follows:

$$w_1 = w_1 - \alpha \cdot dw_1 \qquad \textit{Eq. 4-20}$$

$$w_2 = w_2 - \alpha \cdot dw_2 \qquad \textit{Eq. 4-21}$$

$$b = b - \alpha \cdot db \qquad \textit{Eq. 4-22}$$

All this procedure represents one step of the gradient descent for logistic regression, with respect to a single training example. However, to train the logistic regression model, there will be not just one training example; there will be a training set of m training examples.

When implementing the gradient descent method using the entire training set, the same computations are done but using the vectorized matrices X and Y. In addition, the cost function must be computed using the Eq. 4-9. Hence, the derivative dw_1 would be:

$$\frac{d}{dw_1}J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\frac{d}{dw_1}L\left(\hat{y}^{(i)}, y^{(i)}\right) \qquad \textit{Eq. 4-23}$$

The same operation can be used for the derivatives dw_2 and db, and finally, the gradient descent update the parameters as in the Eq. 4-20, Eq. 4-21 and Eq. 4-22.

4.2. Shallow Neural Networks

4.2.1. Introduction

A neural network can be built by taking many neurons and stacking them together in more intricate structures. These structures are organized into distinct layers of neurons, concatenating the results between layers, allowing NNs to achieve more powerful results.

A simple NN is shown Figure 4-9. This neural network is comprised of the following layers:

- the input layer with three input features
- the hidden layer, which consists in three neurons, where each takes all three features as inputs and sends its output to the next layer
- the output layer, which computes the output $\hat{y}$.

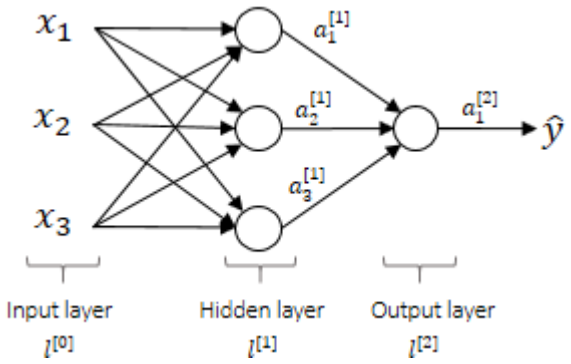


Figure 4-9: 2 layer NN example [9].

In conventional usage, people refer to this particular neural network as a 2-layer neural network, because the input layer does not count as an official layer.

The first layer (layer zero) contains always the input data and in the final layer (output layer) the output of the network is computed. Between these layers, there are the hidden layers, whose number can vary. All these layers are densely connected because all the input features (or neurons in the previous layer) are connected to each neuron in the next layer.

In the Deep Learning community, the common term for neural networks with a single hidden layer (2-layer NNs) is "Shallow Neural Networks", while for neural networks with more hidden layers is "Deep Neural Networks".

As mentioned previously in logistic regression (Figure 4-7), the neuron represents two steps of computation: first, it computes z value using a linear function and then, it computes the activation a as a sigmoid function of z. Accordingly, in a neural network, each neuron computes $z_i^{[l]}$ and $a_i^{[l]}$ values, where the superscript $[l]$ refers to the layer where the computation is done, and the subscript i refers to the neuron in that layer that does the computation (Figure 4-9). These computations are similar to those presented in logistic regression, but the difference is that, whereas in the logistic regression model there is a single z calculation followed by an a calculation, in a neural network, these calculations are done multiple times.

4.2.2. Computing a neural network's output

As said previously, each neuron computes $z_i^{[l]}$ and $a_i^{[l]}$ values. For example, the first node in the hidden layer does the following two steps of computation:

$$z_1^{[1]} = w_1^{[1]T} \cdot x + b_1^{[1]} \qquad \textit{Eq. 4-24}$$

$$a_1^{[1]} = \sigma(z_1^{[1]}) \qquad \textit{Eq. 4-25}$$

In these equations (Eq. 4-24 and Eq. 4-25), because all the quantities are associated with the first node of the first hidden layer, the superscript and the subscript are equal to 1. Therefore, considering the NN in Figure 4-9, the following equations (Eq. 4-26, Eq. 4-27 and Eq. 4-28) show the computations performed by the nodes of the first (and only) hidden layer.

$$z_1^{[1]} = w_1^{[1]T} \cdot x + b_1^{[1]} \qquad a_1^{[1]} = \sigma(z_1^{[1]}) \qquad \textit{Eq. 4-26}$$

$$z_2^{[1]} = w_2^{[1]T} \cdot x + b_2^{[1]} \qquad a_2^{[1]} = \sigma(z_2^{[1]}) \qquad \textit{Eq. 4-27}$$

$$z_3^{[1]} = w_3^{[1]T} \cdot x + b_3^{[1]} \qquad a_1^{[1]} = \sigma(z_3^{[1]}) \qquad \textit{Eq. 4-28}$$

When implementing a NN in deep learning, doing these computations using a for loop is very inefficient when working with processors. Therefore, since deep learning requires a lot of time to train a network, it is necessary to optimize the code avoiding thus de for loops when possible.

The best way to avoid this is to use vectorization. Vectorization tries to convert a stack of equations into a single equation using matrices. One of the general rules in vectorization is that, if there are different nodes in a layer, they are stacked vertically.

First, let's start by showing how to compute z as a vector. Notice that the hidden layer and the output layer will have parameters associated with them. In the case of the hidden layer, it will

have associated with it the parameters $W^{[1]}$ and $b^{[1]}$. Since each neuron has a corresponding parameter vector $w_i^{[l]}$, to obtain the $W^{[1]}$ matrix, these column vectors (from Eq. 4-26, Eq. 4-27 and Eq. 4-28) are transposed and stacked into a matrix as follows:

$$W^{[1]} = \begin{bmatrix} \leftarrow & w_1^{[1]T} & \rightarrow \\ \leftarrow & w_2^{[1]T} & \rightarrow \\ \leftarrow & w_3^{[1]T} & \rightarrow \end{bmatrix}$$

In this example, $W^{[1]}$ will be a 3x3 matrix, being the number of rows equal to the number of nodes in the layer and the number of columns equal to the number of input features. The same is done with $b^{[1]}$, which, in this example, will be a 3x1 vector, being the number of rows the number of nodes in the layer. Once the above equations have been vectorized, the matrices can be multiplied as follows:

$$z^{[1]} = \begin{bmatrix} \leftarrow & w_1^{[1]T} & \rightarrow \\ \leftarrow & w_2^{[1]T} & \rightarrow \\ \leftarrow & w_3^{[1]T} & \rightarrow \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} \cdot x + b_1^{[1]} \\ w_2^{[1]T} \cdot x + b_2^{[1]} \\ w_3^{[1]T} \cdot x + b_3^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \end{bmatrix}$$

The next step is to compute the activation value $a^{[1]}$, being this value the sigmoid function of $z^{[1]}$. In this case, the sigmoid function takes in the three elements of $z^{[1]}$ and applies the sigmoid function element-wise to it. In other words, the sigmoid function is applied on each element of the vector, one by one. As shown in Eq. 4-29, the $a^{[1]}$ value is obtained by stacking together each of the activation values.

$$a^{[1]} = \sigma(z^{[1]}) = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \end{bmatrix} \qquad \textit{Eq. 4-29}$$

The computations of $z^{[l]}$ and $a^{[l]}$ are done in each layer of the neural network. According to the previous example, the values $z^{[1]}$ and $a^{[1]}$ are computed in the hidden layer and the values $z^{[2]}$ and $a^{[2]}$ are computed in the output layer. Therefore, the set of equations necessary to compute the output of the neural network is:

Given input x:

$$z^{[1]} = W^{[1]} \cdot x + b^{[1]} \qquad \textit{Eq. 4-30}$$

$$a^{[1]} = \sigma(z^{[1]}) \qquad \textit{Eq. 4-31}$$

$$z^{[2]} = W^{[2]} \cdot a^{[1]} + b^{[2]} \qquad \textit{Eq. 4-32}$$

$$a^{[2]} = \sigma(z^{[2]}) \qquad \textit{Eq. 4-33}$$

Where the input vector x can be represented by $a^{[0]}$. Notice that the activations $a^{[l]}$ refer to the values that different layers of the neural network are passing on the subsequent layers (Figure 4-9). Finally, the value $a^{[2]}$ is the final output of the neural network and will also be used interchangeably with $\hat{y}$. As shown in Figure 4-10, this value is used to compute the loss in the forward propagation step.

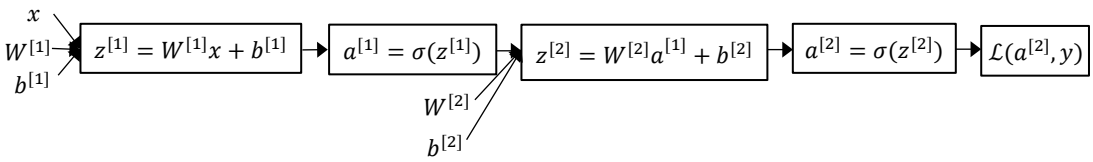


Figure 4-10: Computation graph of the simple NN example [9].

In conclusion, to compute the output of the neural network, z and a calculations are made multiple times; they are performed in each layer by vectorizing the equations of all nodes of that layer. Additionally, as explained for logistic regression, since there is more than one training example, vectorization across multiple training examples will also be required.

4.2.3. Vectorization across multiple samples

In the last section, the equations from Eq. 4-30 to Eq. 4-33, were used to compute the output $\hat{y}$ for a single training example using a NN with a single hidden layer. Now, given this neural network (Figure 4-9) and m training examples, these four equations should be implemented repeatedly for each sample. So, this would consist of implementing the next for loop:

for i=1 to m:

$$z^{[1](i)} = W^{[1]} \cdot x^{(i)} + b^{[1]} \qquad \textit{Eq. 4-34}$$

$$a^{[1](i)} = \sigma(z^{[1](i)}) \qquad \textit{Eq. 4-35}$$

$$z^{[2](i)} = W^{[2]} \cdot a^{[1](i)} + b^{[2]} \qquad \textit{Eq. 4-36}$$

$$a^{[2](i)} = \sigma(z^{[2](i)}) \qquad \textit{Eq. 4-37}$$

Where the superscript (i) refers to the $i\text{-}th$ training example. So they are basically the same four equations as before adding the superscript (i) to all the variables that depend on the training example. But again, vectorized code is used to avoid the for loop. This can be achieved by stacking the training samples (column vectors) into a matrix X as follows:

$$X = \begin{bmatrix} \uparrow & \uparrow & & \uparrow \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ \downarrow & \downarrow & & \downarrow \end{bmatrix}$$

This matrix X is going to contain the input data and will be a (n_x, m) dimensional matrix, where n_x is the number of features of the input vector x. Then the vectorized implementation of the for loop will be:

$$Z^{[1]} = W^{[1]} \cdot X + b^{[1]} \quad \textit{Eq. 4-38}$$

$$A^{[1]} = \sigma(Z^{[1]}) \quad \textit{Eq. 4-39}$$

$$Z^{[2]} = W^{[2]} \cdot A^{[1]} + b^{[2]} \quad \textit{Eq. 4-40}$$

$$A^{[2]} = \sigma(Z^{[2]}) \quad \textit{Eq. 4-41}$$

Note that capital X matrix is obtained by stacking up the lower case vectors $x^{(i)}$ in different columns. Then, by doing the same with the vectors $z^{[1](i)}$ and $a^{[1](i)}$, the matrices $Z^{[1]}$ and $A^{[1]}$ can be obtained:

$$Z^{[1]} = \begin{bmatrix} \uparrow & \uparrow & & \uparrow \\ z^{[1](1)} & z^{[1](2)} & \dots & z^{[1](m)} \\ \downarrow & \downarrow & & \downarrow \end{bmatrix} \qquad A^{[1]} = \begin{bmatrix} \uparrow & \uparrow & & \uparrow \\ a^{[1](1)} & a^{[1](2)} & \dots & a^{[1](m)} \\ \downarrow & \downarrow & & \downarrow \end{bmatrix}$$

Where, horizontally, the matrix $A^{[1]}$ goes over different training examples and vertically, the different indices correspond to different nodes ("hidden units") in the hidden layer. A similar intuition holds true for the matrix $Z^{[1]}$ where, horizontally, the different indices correspond to different training examples, and vertically, they correspond to different input features.

Finally, the same reasoning applies to the second layer, resulting $Z^{[2]}$ and $A^{[2]}$. Notice that the different layers of a neural network are roughly doing the same computation, and even deeper, neural networks will take these two steps and will do them even more times.

4.2.4. Activation functions

Activation functions are used in neural networks to get the output of a neuron (a value) from the linear result (z value). Which activation function to use in the hidden layers is one of the main decisions to take when building a neural network. So far, in this document the sigmoid activation function has been used, but sometimes other choices can work much better.

As it was seen before in the forward propagation step, the sigmoid function was used to compute the activation value a and it was represented by the symbol $\sigma(\,)$, (Eq. 4-4).

In the more general case, a nonlinear function $g(\,)$ will be used to refer to the nonlinear transformation performed in neurons, where $g(\,)$ may be a different function than the sigmoid function.

For example, there is the hyperbolic tangent function (Eq. 4-42). Whereas the sigmoid function goes between 0 and 1, the hyperbolic tangent function goes between -1 and 1 (Figure 4-11).

$$g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \qquad \textit{Eq. 4-42}$$

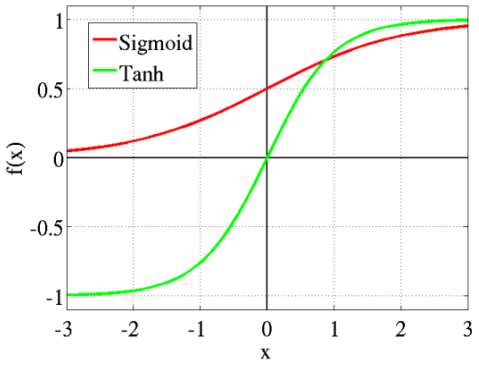


Figure 4-11: Sigmoid function vs tanh function [11]

It turns out that, for the hidden units, the *tanh* function almost always works better than the sigmoid function. This is because the mean of the activations that come out of the hidden layer are closer to 0 and data used in training often has a 0 mean too, and this actually makes learning for the next layer a little bit easier. The only exception is for the output layer because if y is either 0 or 1, then the prediction $\hat{y}$ should be between 0 and 1 instead of between -1 and 1. For this reason, the sigmoid function will be used for the output layer when using binary classification. Notice that the activation functions can be different for different layers.

However, one of the downsides of both the sigmoid function and the hyperbolic tangent function is that if z is either very large or very small, then the slope (or the gradient) of this function ends up being close to zero. And this can slow down gradient descent, since if the gradients are close to zero, the parameters will not update their values. Then, another popular function used in machine learning is the rectified linear unit (ReLU). This function is shown in Figure 4-12 and the formula is (Eq. 4-43):

$$g(z) = \max(0, z) \qquad \textit{Eq. 4-43}$$

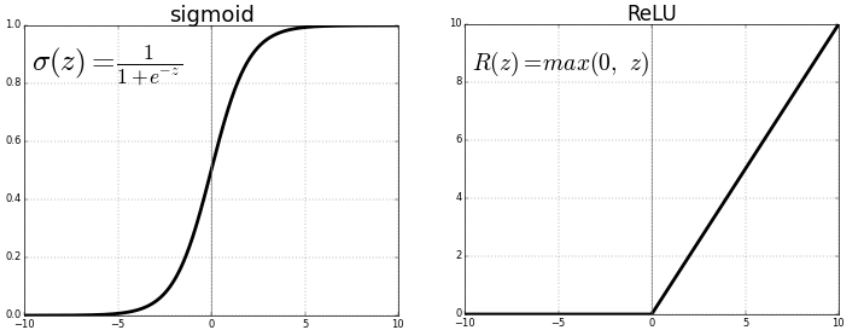


Figure 4-12: Sigmoid function vs ReLU [11]

In the ReLU activation function, the derivative is 1 whenever z is positive, and the derivative is 0, when z is negative. Technically, the derivative when z is exactly 0 is not well defined, but in

practice it is very difficult to get exactly z equal to 0. However, when z is equal to 0, the derivative will be assumed to be either 0 or 1 to ensure proper operation. Nowadays, more and more people are using ReLU as the activation function for the hidden layers.

One disadvantage of the ReLU is that the derivative is equal to zero, when z is negative. Although in practice this works well, there is another version of the ReLU called the Leaky ReLU, which instead of it being 0 when z is negative, it takes a slight slope as shown in Figure 4-13. This usually works better than the ReLU activation function, although it is just not used as much in practice.

$$g(z) = \max(0.01z, z) \qquad \textit{Eq. 4-44}$$

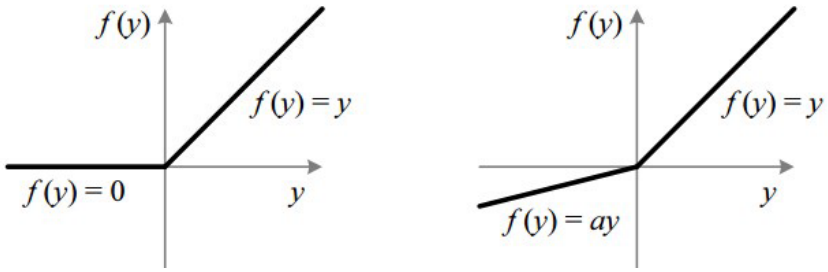


Figure 4-13: ReLU vs Leaky ReLU, where z is represented by y [11]

Notice that the gradient of the Leaky ReLU activation function is 0,01 when z is negative to avoid the null gradient of the ReLU function. This constant can be trained as another parameter of the learning algorithm.

The advantage of both the ReLU and the Leaky ReLU is that for a lot of space of z, the slope of the function is different from 0, which speed up learning. Therefore, in practice, using either of these two activation functions, the neural network will often learn much faster than when using the hyperbolic tangent function or the sigmoid activation function.

In conclusion, the sigmoid function will be used only in the output layer in a binary classification problem. In the other cases, the ReLU or the Leaky ReLU can be used. The most commonly used activation function is ReLU, but good results (or even better results) can also be obtained by using the Leaky ReLU.

Activation functions are required in neural networks and must be nonlinear functions. If linear activation functions ($g(z) = z$) are used, then the neural network is just outputting a linear function of the input, not being able to fit any complex function.

4.2.5. Gradient Descent for neural networks

In this section, Gradient Descent is implemented for a neural network with one hidden layer. This NN will have the following parameters: $W^{[1]}$ which dimensions are ($n^{[1]}, n_x$), $b^{[1]}$ which

is a column vector $(n^{[1]}, 1)$, $W^{[2]}$ which dimensions are $(n^{[2]}, n^{[1]})$ and $b^{[2]}$ which is a real number. Then, the cost function will be:

$$J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) \qquad \textit{Eq. 4-45}$$

When training a neural network, it is important to initialize the parameters randomly rather than with the 0 value. After initializing the parameters, gradient descent is implemented repeatedly to train them though the following loop:

Repeat {

$$Z^{[1]} = W^{[1]} \cdot X + b^{[1]} \qquad \textit{Eq. 4-46}$$

$$A^{[1]} = g^{[1]}(Z^{[1]}) \qquad \textit{Eq. 4-47}$$

$$Z^{[2]} = W^{[2]} \cdot A^{[1]} + b^{[2]} \qquad \textit{Eq. 4-48}$$

$$\hat{Y} = A^{[2]} = g^{[2]}\left(Z^{[2]}\right) = \sigma(Z^{[2]}) \qquad \textit{Eq. 4-49}$$

Compute the cost function J

$$dW^{[1]} = \frac{dJ}{dW^{[1]}}; \quad db^{[1]} = \frac{dJ}{db^{[1]}}; \quad dW^{[2]} = \frac{dJ}{dW^{[2]}}; \quad db^{[2]} = \frac{dJ}{db^{[2]}}$$

$$W^{[1]} = W^{[1]} - \alpha \cdot dW^{[1]} \qquad \textit{Eq. 4-50}$$

$$b^{[1]} = b^{[1]} - \alpha \cdot db^{[1]} \qquad \textit{Eq. 4-51}$$

$$W^{[2]} = W^{[2]} - \alpha \cdot dW^{[2]} \qquad \textit{Eq. 4-52}$$

$$b^{[2]} = b^{[2]} - \alpha \cdot db^{[2]} \qquad \textit{Eq. 4-53}$$

The first four steps in this loop compute the predictions. This involves computing $\hat{Y}$ using the vectorized equations of the forward propagation (equations from Eq. 4-46 to Eq. 4-49). Considering that this is a binary classification problem, the activation function of the output layer should be the sigmoid function (Eq. 4-49).

Then, the cost function is computed using the Eq. 4-45 and the output $\hat{Y}$.

The next step is the backward propagation step, which consists in computing the partial derivatives of the cost with respect to each of the parameters. The following equations are required to compute these derivatives:

$$dZ^{[2]} = A^{[2]} - Y \qquad \textit{Eq. 4-54}$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T} \qquad \textit{Eq. 4-55}$$

$$db^{[2]} = \frac{1}{m}\sum_{i=1}^{n^{[2]}} dz^{[2](i)} \quad \text{Eq. 4-56}$$

$$dZ^{[1]} = W^{[2]T} \cdot dZ^{[2]} * g^{[1]'}(Z^{[1]}) \quad \text{Eq. 4-57}$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T \quad \text{Eq. 4-58}$$

$$db^{[1]} = \frac{1}{m}\sum_{i=1}^{n^{[1]}} dz^{[1](i)} \quad \text{Eq. 4-59}$$

Where $Y = [y^{(1)}\ y^{(2)}\ \dots\ y^{(m)}]$ is vectorized across examples and its dimensions are $(1, m)$. Therefore, this matrix contains the labels for all the m samples stacked horizontally. Notice that the first three equations are similar to those of logistic regression. In Eq. 4-57, $g^{[1]'}(Z^{[1]})$ is the derivative of the activation function used in the hidden layer. Also in this equation, the operand "*" refers to element-wise product.

Finally, at the end of each iteration of Gradient Descent algorithm, the trainable parameters are updated using the previous partial derivatives and the learning rate *α* (Equations from Eq. 4-50 to Eq. 4-53).

All these computations of gradient descent are repeated until the parameters converge.

4.2.6. Random initialization

For logistic regression, it was fine to initialize the weights to zero but, for a neural network, initialize the parameters to zero and then apply gradient descent, it will not work. It turns out initializing the bias term b to zero is acceptable, but initializing W to zero will be a problem.

In a NN, if all weights are initialized with zeros, then all the hidden units are completely identical because all of them are computing exactly the same function. So, the activation values will be the same for each hidden unit and will have the same influence on the output. If every neuron computes the same output, they will also compute the same gradients during the backward pass and undergo the same parameter update [13]. Thus, all neurons will evolve symmetrically throughout training, preventing different neurons from learning different things [14].

The solution to break this symmetry is to initialize the weights randomly. Specifically, nodes that are side-by-side in a hidden layer connected to the same inputs must have different weights so that the learning algorithm updates the weights differently for each unit [15].

Consequently, weights are initialized to very small random values and biases are initialized to zero since the symmetry breaking problem no longer exists. It turns out that initializing the weights to a very small random values is preferable because, if the weights are too large, then values of z will be either very large or very small, and will be more likely to end up at the flat parts of the sigmoid or the *tanh* function where the gradient is very small. Therefore, weights that are too large will slow down the gradient descent and, thus, also the learning.

4.3. Deep Neural Networks

4.3.1. Introduction

In the previous sections, the logistic regression and a neural network with a single hidden layer have been explained. Whereas these functions are known as “shallow” models, a neural network with more hidden layers is known as a “deep” model. The Figure 4-14 shows a deep neural network with 3 hidden layers, called a 4-Layer neural network.

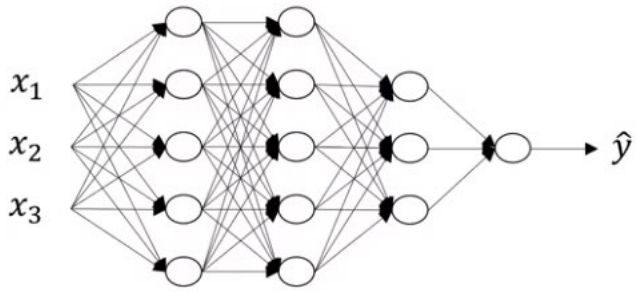


Figure 4-14: 4-Layer neural network

The logistic regression model is a very shallow model, whereas the model in Figure 4-14 is a deeper model. Then, the 2-Layer NN (with one hidden layer) is still quite shallow, but not as shallow as logistic regression.

For any given problem, it might be hard to predict in advance exactly how deep the neural network should be. So it seems reasonable to test the logistic regression, then a neural network with one hidden layer, then with two hidden layers, and take the number of hidden layers as another hyper-parameter to tune in order to try to find the right depth for the NN.

The notation used to describe deep neural networks will be:

- L: number of layers in the network
- $n^{[l]}$: number of units in layer l
- $a^{[l]}$: activation values in layer l and $a^{[l]} = g^{[l]}(z^{[l]})$, where $g^{[l]}$ is the activation function used in layer l and $z^{[l]}$ is the linear result in layer l.
- $W^{[l]}$ and $b^{[l]}$: weights and biases in layer l, used to compute the value of $z^{[l]}$.

Finally, the input features are represented by x but, they are also the activations of layer zero $a^{[0]}$. In addition, the activation value of the output layer $a^{[L]}$ is equal to the prediction $\hat{y}$.

4.3.2. Why deep representations?

Deep neural networks (DNNs) work well for a lot of problems. An example is used below to explain this statement: the face recognition or face detection system.

In a face recognition or a face detection system (Figure 4-15), the input data will be a picture of a face. Then, the first layer of the neural network can be considered as a feature detector or an edge detector. The image on the left shows the possible calculation that the first layer would make with 20 hidden units. A hidden unit (represented by one of these small square boxes) is trying to figure out where are the edges of that orientation in the picture.

Then, the second layer will possibly take the detected edges and group them together to detect different parts of faces. For example, one neuron will try to find an eye, while another neuron will try to find a part of the nose.

Finally, by putting together different parts of the faces, the next layer will try to recognize or detect different types of faces.

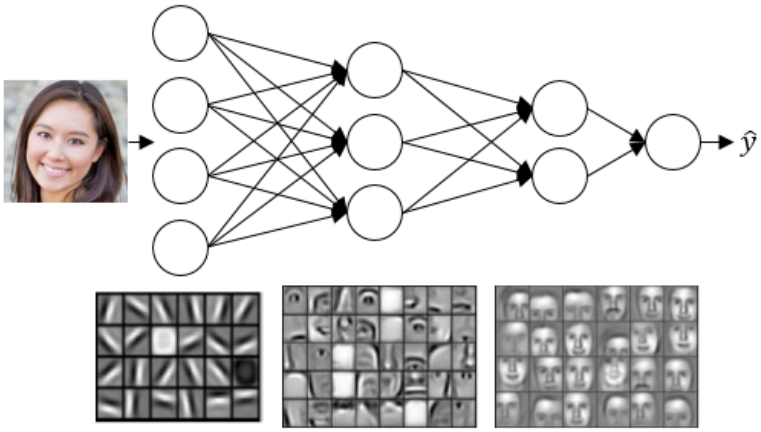


Figure 4-15: Face recognition or face detection system [9].

As can be seen, the earlier layers of the neural network can learn simple functions (such as detecting edges) and then, composing them together, the later layers can learn increasingly complex functions. To do that, edge detectors examine relatively small areas of an image, while facial detectors examine the larger areas. Is not really known with certainty what the network learns, but is known that it goes from simple to complex functions.

Some people like to make an analogy between DNNs and the human brain, since neuroscientists believe that the human brain also starts by detecting simple things like edges in what the eyes see and then, put them together to detect more complex things.

The other reason about why DNNs seem to work well is the following:

There are functions that can be computed with "small" L-layer DNN, which is a neural network with multiple hidden layers but with a relatively small number of hidden units. The higher the number of layers, the more complex can be the functions that the network calculates without incurring a very high computational cost. If the same functions were computed with a shallower NN (with few hidden layers), then they might require exponentially more hidden units in each layer to compute them. Then, having more hidden units would entail a higher computational cost of the operations performed.

4.3.3. Forward propagation in a deep neural network

The forward propagation in a deep neural network is performed using the following equations in each layer:

$$Z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]} \qquad \textit{Eq. 4-60}$$

Where $W^{[l]}$ and $b^{[l]}$ are the parameters that affect the activations in layer l. And then:

$$A^{[l]} = g^{[l]}(Z^{[l]}) \qquad \textit{Eq. 4-61}$$

And finally, $\hat{Y} = A^{[L]}$ where $\hat{Y}$ has the predictions on all training examples stacked horizontally.

In deep neural networks, a for loop is needed to compute the activations for each layer, ranging from the first layer to layer L (the final layer). As previously said, it is preferable to explicitly avoid for loops when implementing neural networks, but here there is no way to avoid it. Therefore, when implementing forward propagation, it is acceptable to have a for loop.

Therefore, these are the vectorized general equations for forward propagation in a deep neural network. Note that these equations are similar to those seen in the previous sections with shallow neural networks, but here, they will be repeated more times.

4.3.4. Backward propagation in a deep neural network

The backward propagation step in a deep neural network is performed using the following equations in each layer:

$$dZ^{[l]} = dA^{[l]} * g^{[l]'}(Z^{[l]}) \qquad \textit{Eq. 4-62}$$

$$dW^{[l]} = \frac{1}{m} \cdot dZ^{[l]} \cdot A^{[l-1]T} \qquad \textit{Eq. 4-63}$$

$$db^{[l]} = \frac{1}{m} \cdot \sum_{i=1}^{m} dz^{[l](i)} \qquad \textit{Eq. 4-64}$$

$$dA^{[l-1]} = W^{[l]T} \cdot dZ^{[l]} \qquad \textit{Eq. 4-65}$$

Where the operand "*" denotes element-wise multiplication.

It turns out that, combining the Eq. 4-62 and Eq. 4-65, the equation to compute $dZ^{[l]}$ (*Eq. 4-66*) will be the same as the one seen in the previous sections for a shallow neural network.

$$dZ^{[l]} = W^{[l+1]T} \cdot dZ^{[l+1]} * g^{[l]'}(Z^{[l]}) \qquad \textit{Eq. 4-66}$$

Notice that these are the vectorized general equations for backward propagation in a deep neural network.

4.3.5. Building blocks representation

The forward and the backward propagation steps seen in the previous sections can be represented as two basic building blocks. Taking the vectorized equations for any layer l, the forward and backward propagation blocks are those shown in Figure 4-16.

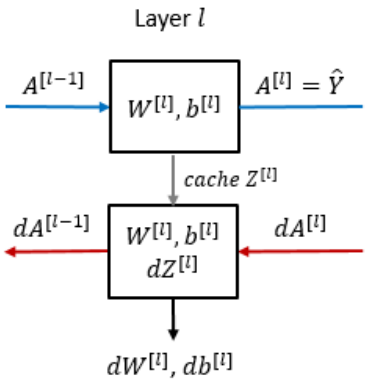


Figure 4-16: Forward and backward functions for a generic layer l. The top building block is for the forward propagation step and the bottom building block is for the backward propagation step [9].

As can be seen in Figure 4-16, the input of the forward function is the activation of the previous layer $A^{[l-l]}$ and the output is the activation of the l layer $A^{[l]}$. In order to perform this computation, the function uses the parameters of the layer $W^{[l]}$and $b^{[l]}$. In addition, this function also outputs a cache. A cache is a variable that stores useful values that are computed in forward propagation in order to reuse them later in backward propagation. The cache saves work, since otherwise these values would need to be recalculated. Normally, it contains $Z^{[l]}$, as well as $A^{[l-1]}$, and the parameters $W^{[l]}$ and $b^{[l]}$ for each layer.

In the figure, the blue arrows show the direction of the forward propagation and the gray arrow show the direction of the cache, which is entering the backward propagation building block.

On the other hand, the input of the backward function is the derivative of the loss function with respect to the activation of the l layer, represented by $dA^{[l]}$ and the output is the derivative of the loss function with respect to the activation of the previous layer and it is represented by $dA^{[l-1]}$. According to what was seen in the previous section, using the input $dA^{[l]}$ and the $Z^{[l]}$ value stored, the $dZ^{[l]}$ value can be computed. Then, also using the values in the cache, the backward function can also output the derivatives $dW^{[l]}$ and $db^{[l]}$. These derivatives are used when updating the parameters $W^{[l]}$and $b^{[l]}$ in order to implement gradient descent for learning. In Figure 4-16, the red arrows show the direction of the backward propagation.

These two basic components are the key components necessary to implement a deep neural network. By concatenating these components, the deep neural network can be represented as follows:

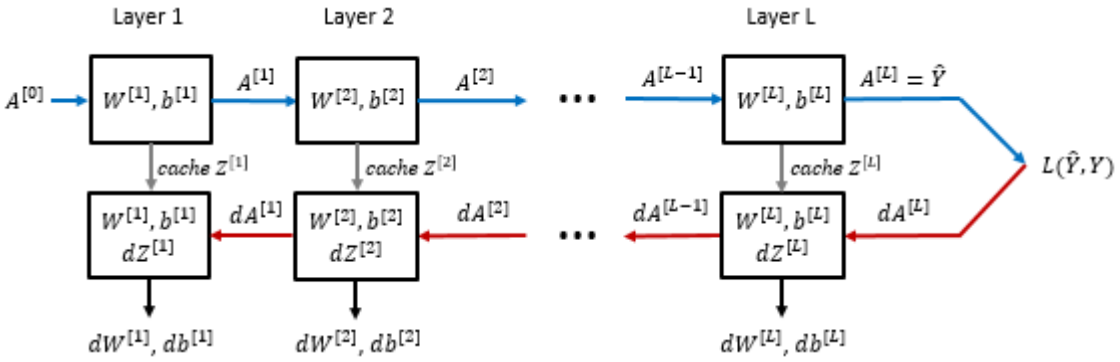


Figure 4-17: Building blocks representation or deep neural networks [9].

As shown in Figure 4-17, in each layer there is a forward propagation step and the corresponding backward propagation step, and there is also a cache to pass information from one to the other.

So, an iteration of training through a neural network involves: starting with $A^{[0]}$, which is the input X, and going through forward propagation until computing $A^{[L]}$ which is equal to $\hat{Y}$. Then, using the loss function, calculating the derivative term $dA^{[L]}$ (*Eq. 4-68*). Then, going through backward propagation computing the derivatives and finally, updating the parameters in each layer using the derivative terms $dW^{[l]}$ and $db^{[l]}$.

Whereas the forward propagation step is initialized with the input data X, the backward propagation step is initialized with the derivative $dA^{[L]}$, which is the derivative of the loss function with respect to the output $A^{[L]}$ (or, $\hat{Y}$). As seen before, when doing binary classification, this derivative for a single training example is as follows:

$$da^{[L]} = -\frac{y}{a^{[L]}} + \frac{(1-y)}{(1-a^{[L]})} \qquad \textit{Eq. 4-67}$$

Where y is the label.

Then, the vectorized implementation that should be used for the final layer is the following:

$$dA^{[L]} = \left(-\frac{y^{(1)}}{a^{[L](1)}} + \frac{(1-y^{(1)})}{(1-a^{[L](1)})} \quad -\frac{y^{(2)}}{a^{[L](2)}} + \frac{(1-y^{(2)})}{(1-a^{[L](2)})} \quad \dots \quad -\frac{y^{(m)}}{a^{[L](m)}} + \frac{(1-y^{(m)})}{(1-a^{[L](m)})}\right) \qquad \textit{Eq. 4-68}$$

Actually, there is one more output to compute, $dA^{[0]}$, but this derivative with respect to the input features is not useful at least to train the weights of these supervised neural networks.

Finally, this is one iteration of gradient descent, so this procedure is repeated until the parameters converge.

4.3.6. Parameters and hyper-parameters

When training deep neural networks, being efficient in network development requires not only organizing the parameters well but also the hyper-parameters.

The parameters of the model are W and b. These are the parameters that the network learns.

In addition to W and b parameters, the learning algorithm requires other parameters called "hyper-parameters". For example:

- Learning rate α
- Number of iterations
- Number of layers L
- Number of hidden units in each layer $n^{[1]}, n^{[2]}, \dots, n^{[L]}$
- Activation function used in each layer $g^{[1]}, g^{[2]}, \dots, g^{[L]}$ (sigmoid, ReLU, tanh…)

And there are many more hyper-parameters. These hyper-parameters are the variables that determine the network structure and they determine how the network should be trained.

The network parameters are learned from data during the training, whereas the hyper-parameters are manually set earlier and are used to control the ultimate parameters W and b.

When training a deep neural network, there may be a lot of possible settings for the hyper-parameters that should be tested. When starting on a new application, it may be difficult to know in advance exactly what is the best value of the hyper-parameters, and for this reason, what people have to do is to try out different values going around the circle shown in Figure 4-18, and see how they work. So, applying deep learning today is an empirical and iterative process, where there is the idea, then the coding and finally, the experiment to test the idea.

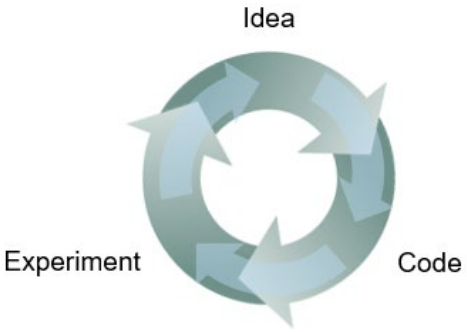


Figure 4-18: Empirical and iterative process [9]

After doing all the iterations, the value of the hyper-parameter that would be good to take will be the value that provides fast learning and allows convergence to a lower cost function.

4.4. Convolutional neural networks

4.4.1. Computer vision and convolutional neural networks

Computer vision is one of the areas that has been advancing rapidly thanks to deep learning. These rapid advances in computer vision are allowing new applications to be seen, although they were impossible a few years ago.

For example, deep learning computer vision is now helping self-driving cars figure out where the other cars and pedestrians are so as to avoid them. Also, it is making face recognition work much better than before, and nowadays, it is already possible to unlock a phone or even a door using just the face. And also, there are many mobile phone apps that use deep learning to show users the images that may be most attractive, beautiful or relevant to them.

Here are some examples of computer vision problems:

- Image classification problem:

  In an image classification problem, the model takes an image as input and, specifically for the cat classifier (Figure 4-19), tries to find out if there is a cat in the image or not.

Figure 4-19: Cat classifier example [9].

- Object detection problem:

  In an object detection problem, it is generally important not only to discover that there are objects in the image, but also to figure out the their positions in the image; in Figure 4-20 their positions are indicated by rectangular bounding boxes in red.

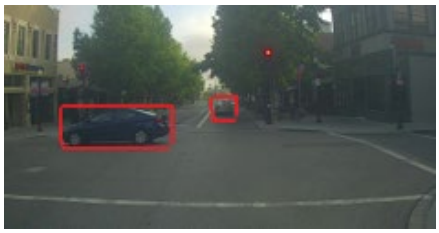


Figure 4-20: Object detector for an autonomous car [9]

One of the challenges of computer vision problems is that network inputs can be really big.

In Figure 4-21, the image on the left is a low resolution image which dimensions are 64x64x3 (it has the three RGB channels). This results in an input vector x with dimension 12288, which is not so bad. But a 64x64 image is actually a very small image.

On the other hand, the image on the right is a higher resolution image which dimensions are 1000x1000x3. In this case, there are 3 million features and that means the input vector x will be 3 million dimensional. Then, if there are 1000 hidden units in the first layer and, as until now, a standard neural network is used, then the matrix $W^{[1]}$ will be a 1000 by 3 million dimensional matrix. This means that this matrix contains 3 billion parameters, which is a very large number. With so many parameters, the computational and memory requirements to train the neural network are a bit infeasible.

Figure 4-21: Two cat images with different resolution [9]

But for computer vision applications, deep learning algorithms should be able to use both small images and large images. To do that, the convolutional operation need to be implemented. This operation is one of the fundamental building blocks of convolutional neural networks.

Convolutional neural networks (CNN) have provided many solutions to the field of computer vision, as they work exceptionally well when processing unstructured data (images or videos), such as in image classification and object detection tasks.

The emergence of powerful and versatile deep learning frameworks in recent years enabled the implementation of convolutional layers, a very simple task, often achievable in a single line of code [16].

4.4.2. Convolutional operation

In convolutional neural networks (CNN), a convolution can be expressed as a tool that creates a feature map from the input data using a filter. Therefore, the feature map is the output of one filter applied to the result of the previous layer and is called “feature map” because it maps the input data to the feature space. This concept will be explained in more detail below.

To do the convolutional operation, there will be the input matrix and a filter (or kernel).

The input matrix can be an image or another feature map resulting from the convolutional operation of the previous layer. On the other hand, the filter is a small matrix (3x3, 5x5) which contains values that are known as weights. In convolutional operation, the filter is convolved with the input matrix. In this example (Figure 4-22), the input matrix is a 5x5 grayscale image and the filter is a 3x3 matrix.

The output (feature map) of this convolutional operation will be a 3x3 matrix. The way to compute this output is as follows:

The upper left element of this output matrix is computed by first taking the filter and superimposing it on the upper left of the original input image as shown in Figure 4-22. Then, performing the element-wise product and finally, adding up all the resulting numbers.

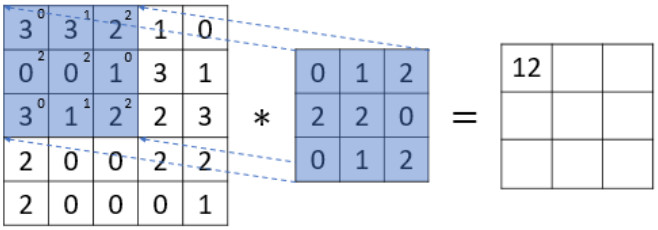


Figure 4-22: Operation to get the first element in the first row

Next, to compute the second element, the filter is taken and shifted one step to the right (Figure 4-23). Here, the same element-wise product is done and then, the addition.

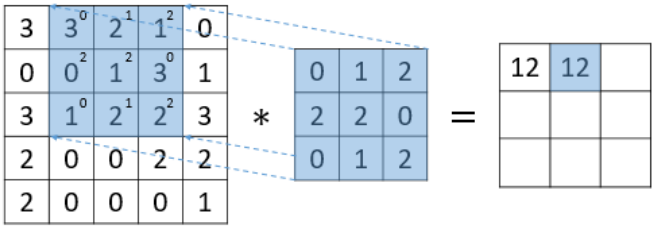


Figure 4-23: Operation to get the second element in the first row

In order to get the element in the next row, the filter is shifted down one position, and the element-wise product and the addition exercise are performed again (Figure 4-24).

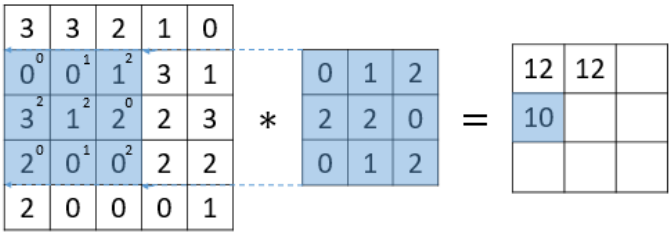


Figure 4-24: Operation to get the first element in the second row

Finally, the rest of the elements of the matrix are computed by repeating the same steps (Figure 4-25).

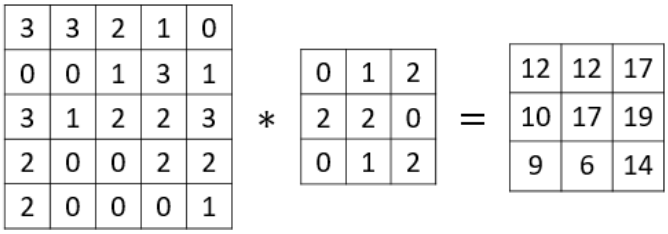


Figure 4-25: Result of the convolutional operation

Edge detection example:

In this section, the edge detection example is used to understand the concept of a feature map, as it makes it easier to see the result of the convolutional operation graphically. Furthermore, this concept of edge detection had already appeared to explain that the first layers of the neural network are responsible for detecting simple features such as edges.

In this example, the input image is a 6x6 grayscale image where the left half of the image is 10 and the right half is zero (Figure 4-26). Plotted as a picture, the 10s give brighter pixel intensive values and the zeros give darker pixel intensive values, so the left half of the image is white and the right half of the image is gray. As can be seen, there is clearly a very strong vertical edge in the middle of the image as it transitions from white to a darker color.

The filter used to detect vertical edges is a 3x3 matrix where there are brighter pixels on the left, zeros in the middle and darker pixels on the right. Then, the feature map resulting from the convolution of the image with the filter is the one shown on the right in the Figure 4-26. As shown in the figure, in the resulting image there is a brighter region in the middle that corresponds to the vertical edge detected in the middle of the 6x6 image.

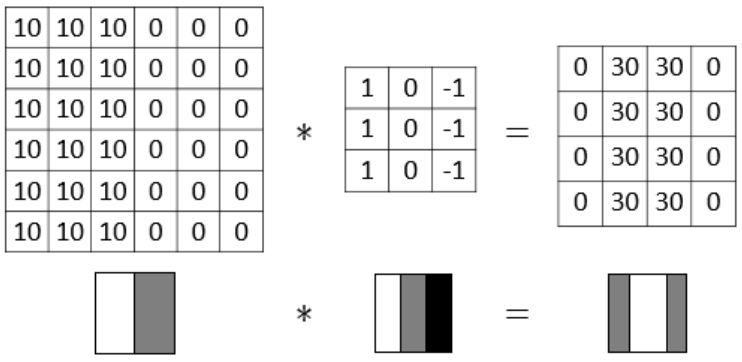


Figure 4-26: Vertical edge detection example [9]

Therefore, this output matrix is called a feature map because it is a map that indicates where a certain feature is located in the image. A high number means that a certain feature was found, for example, a vertical edge.

In this case, the dimensions seem to be wrong since the detected edge looks really thick. This is because small images are being used, but if larger images are used then this does a pretty good job of detecting the vertical edges in the image.

In deep learning models different filters can be used to detect edges. One of the most powerful ideas in computer vision is that these nine numbers of the filter can be treated as parameters (Figure 4-27), which can be learned in the backward propagation step.

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

Figure 4-27: Learnable filter

Therefore, the goal is to learn these parameters to obtain a good edge detector. And the backward propagation function can choose to learn a vertical or horizontal edge detector, or it is more likely to learn something different that captures the statistics of the data even better than any of the hand-coded filters. And instead of just vertical and horizontal edges, maybe it can learn to detect edges that are at 45 degrees or at 70 degrees or at whatever orientation it chooses. And so, by allowing all of these numbers to be parameters and automatically learning them from data, neural networks can learn features more robustly than computer vision researchers are generally able to code up by hand.

4.4.3. Padding

One modification to the basic convolutional operation that can be used is padding.

As mentioned in the previous section, if a 6x6 image is convolved with a 3x3 filter, the result will be a 4x4 output image. This is because the possible positions for the 3x3 filter to fit in the 6x6 image are those in the 4x4 region in the center of the image. Accordingly, if there is an $n\ x\ n$ image and a $f\ x\ f$ filter, the dimension of the output will be: $n - f + 1$ x $n - f + 1$.

There are two disadvantages:

- Shrinking output: As each time the convolutional operation is applied, the image shrinks, so this can only be applied a certain number of times before the image becomes too small. This becomes a problem when building deep neural networks, because if there are many layers, the image will shrink a little on each layer, and then, after a hundred layers, the image will be very small.
- Throwing away a lot of the information from the edges of the image: while the pixel in the corner is used only once to compute the output, the pixel in the middle is used more times (Figure 4-28). Therefore, very few values in the next layer would be

affected by the pixels at the corners or edges of the original image, so a lot of information from the edge of the image is wasted.

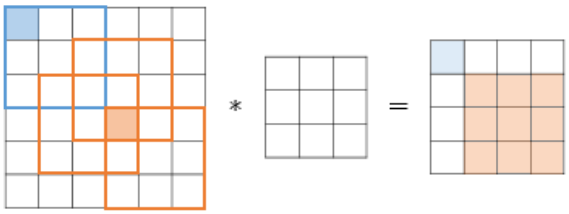


Figure 4-28: Difference of times a pixel in the center of the image and a pixel in the corner are used.

To solve both problems, before applying the convolutional operation, image padding can be used. Image padding introduces new pixels around the edges of an image. By convention, the image is padded with zeros.

Taking the example in Figure 4-28, the original image is padded with an additional one-pixel border around the edges (Figure 4-29). So instead of having a 6x6 input image this is 8x8 and after the convolutional operation, the output is a 6x6 image instead of a 4x4 image. Therefore, when padding is used, the original input size is preserved in the output image.

If “p” is the amount of padding, the output size becomes:

$$(n + 2p - f + 1\ ,\ n + 2p - f + 1)$$

And for the output size to be the same as the original size, the following equality must be fulfilled:

$$p = \frac{f - 1}{2}$$

Where “f” is usually odd.

In this example, “p” is equal to one, because there is an extra boarder of one pixel.

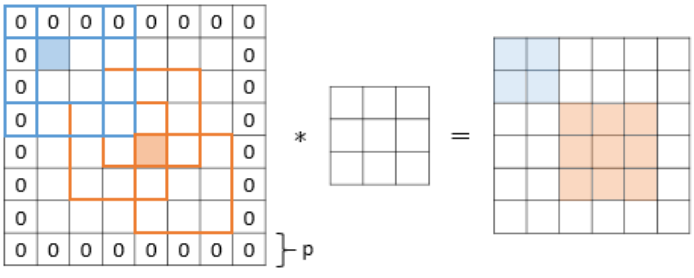


Figure 4-29: Convolutional operation with padding image

As can be seen in Figure 4-29, the use of padding solves both problems mentioned above. First, using a suitable padding according to the filter, the output size can be the same as the input size, avoiding shrinking the image. And second, the pixels at the edges are used

more times, that is, they have more influence on the output and the information from the edges is not wasted. For example, here, the blue pixel in the corner influences 4 cells of the output instead of just one.

4.4.4. Strided convolutions

Strided convolutions are another modification to the basic convolutional operations. The stride "s" governs how many cells the filter is moved in the input to calculate the next cell in the output. A stride of 1 acts as a standard convolution ("non-strided").

In the example in Figure 4-30, the convolutional operation is done with a stride of 2. This means moving the filter two positions to the right or down, instead of just one.

To compute each cell in the output matrix, the same operation is performed as before: first, take the element-wise product and then add the nine numbers. The difference is in how the filter is moved over the original image.

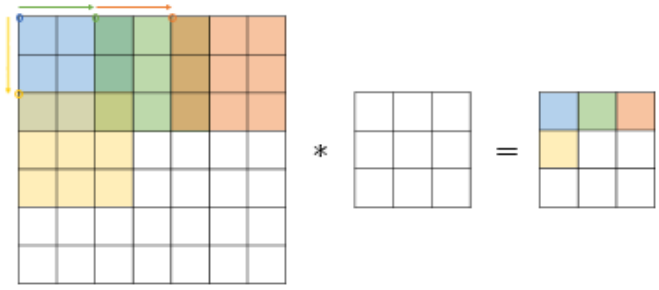


Figure 4-30: Convolutional operation with stride s=2

In this example, the 7x7 input image is convolved with a 3x3 filter and outputs a 3x3 image. Consequently, the output dimensions turn to be governed by the following formula:

$$\left(\frac{n+2p-f}{s}+1,\ \ \frac{n+2p-f}{s}+1\right)$$

Where, n is the input size, f is the filter size, p is the padding and s the stride.

If this fraction is not an integer, the result is rounded down. The way this is implemented is that the convolution is performed only if the filter is completely contained within the image, and if part of the filter hangs outside, that calculation is not done.

As can be seen, using strided convolutions the output has a lower size as the input. This is commonly required in CNN, where the size of spatial dimensions is reduced when increasing the number of channels. Besides, programmers increase the stride in order to save space or reduce the calculation time.

4.4.5. Convolutions over volumes

In the previous sections, the convolutions were implemented over grayscale (2D) images. In this section, the convolutional operation is implemented over volumes.

In this case, the input image is a 3D matrix, where the first dimension is the height of the image, the second is the width and the third is the number of channels. For example, in RGB images, the third dimension will be equal to 3, which corresponds to the three color channels.

In order to perform a convolution over a volume, a three-dimensional filter must be used. Similarly, the filter has a height, a width and a number of channels. An important thing to note is that the number of channels in the input image must match the number of channels in the filter. In the case of using an RGB image as input volume, the filter will also have 3 channels.

Figure 4-31 shows a convolution over an RGB image. So instead of having a 6x6 image, it will be a 6x6x3 image. Considering that there is no padding and that the stride is equal to 1, if the input image is convolved with the 3D filter, the output image will be a 4x4 matrix.

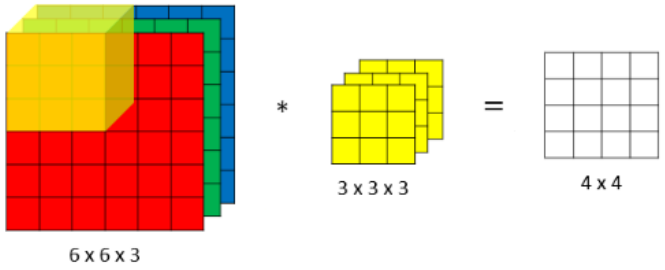


Figure 4-31: Convolution over a 3D volume [9]

To compute the output of this convolutional operation, the filter is taken and, first, it is placed in the upper left position of the input image. In this case, the filter has 27 parameters and each of these 27 numbers are multiplied with the corresponding numbers of the red, green and blue channels of the image (the numbers that are covered by the yellow cube shown in Figure 4-31). Then, those 27 numbers resulting from the element-wise multiplication are added up to give the upper left number of the output matrix.

The rest of the numbers in the output matrix are computed in a similar way to the 2D case, sliding the filter one position to the right or down and in each position, doing the 27 multiplications and adding the 27 numbers.

Multiple filters

If different features are required to be detected, multiple filters can be used at the same time.

In Figure 4-32, the previous example is taken but now, two different filters are used; for example, they could be a vertical edge detector and a horizontal edge detector. Now, when

convolving the RGB image with the first filter, the result will be a 4x4 feature map and when convolving the RGB image with the second filter, the result will be different 4x4 feature map. Then, by stacking these two feature maps together, the result will be a 4x4x2 volume. Notice that the third dimension of the output volume represents the number of filters that have been used.

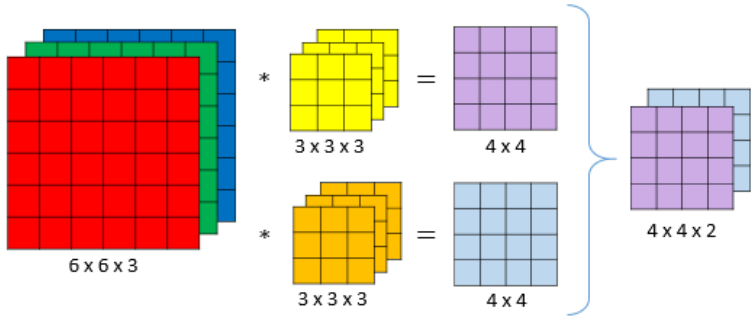


Figure 4-32: Convolution over a 3D volume with 2 different filters [9]

Finally, if the input image dimensions are $n\ x\ n\ x\ n_c$, and the filter dimensions are $f\ x\ f\ x\ n_c$, where, by convention, they both have the same number of channels, then the output volume will have the following dimensions:

$$n - f + 1 \quad x \quad n - f + 1 \quad x \quad n_c'$$

Where n_c' is the number of filters that have been applied. As can be seen in Figure 4-32, the number of filters used will result in same number of feature maps.

Again, these dimensions are valid for zero padding and stride equal to 1. If a different stride or padding is used, it will affect the output dimensions in the same way as seen in the previous sections.

The idea of convolution over volumes turns out to be really powerful. First, because normally images are RGB images and this allows the neural network to operate directly on these images. And second, but even more important, multiple filters allow the NN to detect different features (2, 10, 128 or several hundreds of different features) at the same time.

4.4.6. Layers of a CNN

Convolutional operations are one of the fundamental building blocks of CNNs. The operation explained in the last section can be implemented, with some modifications, as a layer (convolutional layer) in a neural network in order to create a CNN.

Although it is possible to design a fairly good neural network using only convolutional layers (CONV), most neural network architectures will also have pooling layers (POOL) and fully connected layers (FC). Figure 4-33, shows an example of a CNN using these three types of layers.

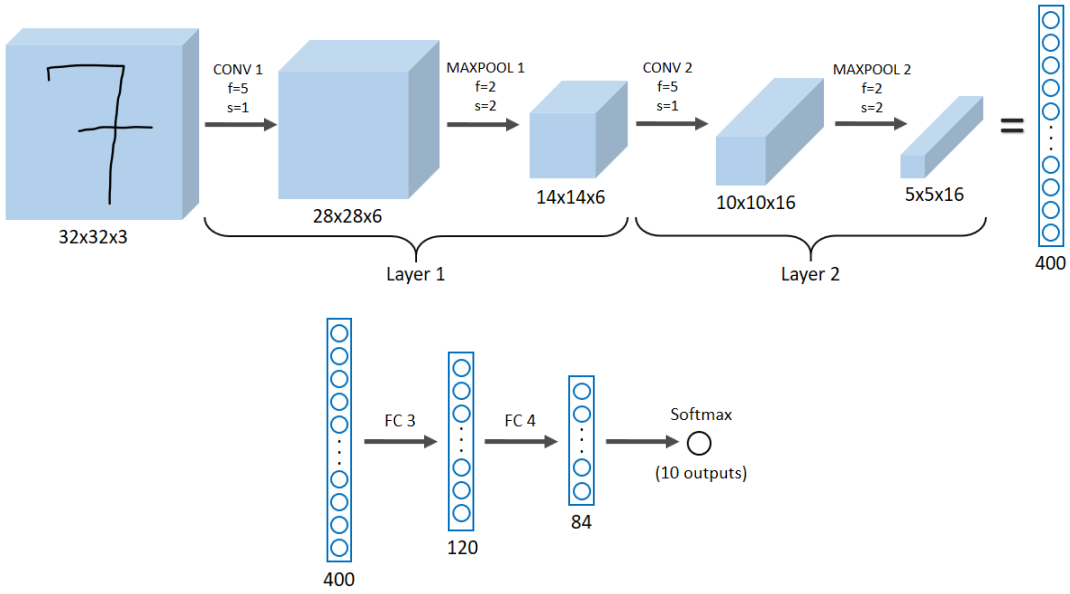


Figure 4-33: CNN example [9]

4.4.6.1. Convolutional layer (CONV)

Convolutional layers in a CNN systematically apply learned filters to input images in order to create feature maps that summarize the presence of those features in the input [17]. To detect different features, multiple filters are used in a convolutional layer.

The modification necessary to turn the operation of the previous section into a convolutional layer consists of applying a bias term and then, a non-linear function (Figure 4-34).

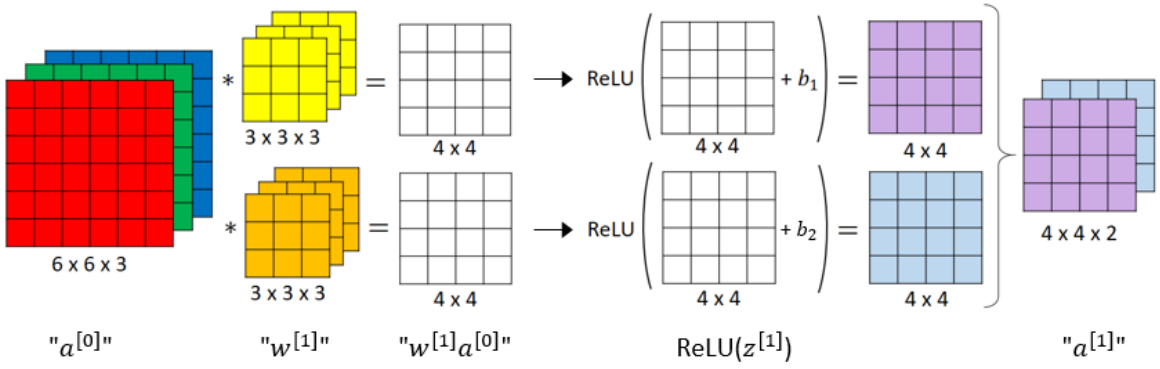


Figure 4-34: Computations for a convolutional neural network layer [9]

First, following the example above, a bias term is added to each element of the 4x4 outputs. As previously stated, the filter contains the parameters w and the bias term is a real number. Each real number (b_1 and b_2) is added to each element of the corresponding 4x4 matrix.

Then, a non-linear function is applied to both linear outputs. In this example, the ReLU function is applied. Finally, after applying the bias and the non-linearity, two 4x4 outputs will be obtained that, when stacked, will result in a 4x4x2 volume.

The computation shown in Figure 4-34 is performed in one CNN layer. As a reminder, the forward propagation equations used in layer l are as follows:

$$z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]} \quad \textit{Eq. 4-69}$$

$$a^{[l]} = g^{[l]}(z^{[l]}) \quad \textit{Eq. 4-70}$$

In this case, the 6x6x3 input image is the activation of the previous layer $a^{[0]}$ and the filters play a role similar to $w^{[1]}$. In this case, the convolutional operation is used to compute the linear function ($w^{[1]}a^{[0]}$) to obtain the 4x4 matrices. Adding bias to these matrices gives the linear result, which plays a role similar to $z^{[1]}$. Finally, by applying the non-linear function, the activation $a^{[1]}$ is obtained.

The output $a^{[1]}$, is comprised of different feature maps, one for each filter used. A feature map can be also called an activation map, hence the $a^{[1]}$ notation.

As explained above, the reason why CNNs are used instead of conventional NNs is related to the number of parameters, since when using NNs with very large images, the number of parameters becomes extremely large. Instead, the advantage of using a CNN is that the number of parameters does not depend on the size of the input image.

Finally, the notation for a convolutional layer l and the dimensions of the different elements are detailed below.

When designing a convolutional layer, the hyper-parameters to be determined are:

- $f^{[l]}$: filter size
- $p^{[l]}$: padding
- $s^{[l]}$: stride
- $n_c{}^{[l]}$: number of filters

The input image in layer l will have the following dimensions:

$$n_H^{[l-1]} \; x \; n_W^{[l-1]} \; x \; n_c^{[l-1]}$$

Where $n_c^{[l-1]}$ is the number of channels (or what is the same, the number of filters) in the previous layer. The subscripts “H” and “W” are used to denote the height and width of the image in case their values differ.

On the other hand, the output volume size will be:

$$n_H^{[l]} \; x \; n_W^{[l]} \; x \; n_c^{[l]}$$

Where the number of channels in the output volume is equal to the number of filters used in this layer. The other two dimensions, $n_H^{[l]}$ and $n_W^{[l]}$, can be calculated using the Eq. 4-71.

$$n_H^{[l]} = \left\lfloor \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor \qquad \textit{Eq. 4-71}$$

And technically this is also true for the weight. The symbol "⌊ ⌋" indicates that the result should be rounded down.

Then, each filter is going to be a $f^{[l]}\ x\ f^{[l]}\ x\ n_c^{[l-1]}$ volume, since the number of channels in the input volume must match the number of channels in the filter. As the filters contain the weights, then the weight matrix $W^{[l]}$ will actually be the result of putting all the filters together. Accordingly, the dimension of this matrix will be the dimension of a filter multiplied by the number of filters:

$$f^{[l]}\ x\ f^{[l]}\ x\ n_c^{[l-1]}\ x\ n_c^{[l]}$$

On the other hand, the bias term is a real number and there is one bias for each filter. Therefore, the bias is just a $n_c^{[l]}$ vector.

Lastly, the activation of this layer will have the same dimensions as the output of the layer. However, when using a vectorized implementation, the activation $A^{[l]}$ will have the following dimensions:

$$m\ x\ n_H^{[l]}\ x\ n_W^{[l]}\ x\ n_c^{[l]}$$

4.4.6.2. Pooling layer (POOL)

Pooling layers are another type of layer commonly used in CNNs. It is common to insert a pooling layer between successive convolutional layers in a CNN architecture. These layers are often used to reduce the size of the representation of the data to speed up the computation, as well as to make some of the features detected by the CNNs somewhat more robust.

In a pooling layer, the first thing to do is define a filter and the type of operation to perform, either the max pooling or the average pooling. An interesting property of pooling is that it has a set of hyper-parameters but has no parameters to learn. The hyper-parameters are:

- $f^{[l]}$: filter size
- $s^{[l]}$: stride
- Max pooling or Average pooling

Similar to convolutional layers, the filter size and stride are hyper-parameters. Padding could also be added as a hyper-parameter, but is not normally used ($p^{[l]} = 0$).

Once the hyper-parameters are chosen, the input feature map is divided into different regions of the filter size. Then, in each region the same operation is performed. Since hyper-parameters are fixed and this pooling layer has no parameters, this operation is a fixed function. That is, since there are no parameters to learn, the gradient descent will not change anything during the backward propagation step.

Max Pooling:

There is a type of pooling called max pooling or maximum pooling that consists of taking the maximum value of each region of the feature map.

Figure 4-35 shows an example, where the input is a 4x4 matrix and the filter size and the stride are equal to 2. In this example, since the filter is a 2x2 matrix, the input matrix is divided in four 2x2 regions. Then, each of the numbers in the output matrix will be the max of the numbers in the corresponding reshaped region. Therefore, the output is a 2x2 matrix.

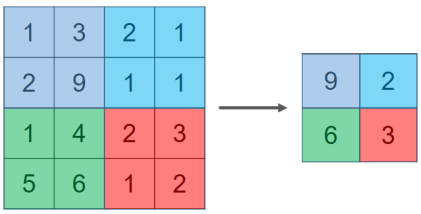


Figure 4-35: Max pooling example with f=2 and s=2

In this case, the results are downsampled feature maps that highlight the most important feature in the region [17].

Average pooling:

The other type of pooling is the average pooling, which consists of calculating the average value for each region of the feature map.

In the example in Figure 4-36, the input feature map and the hyper-parameter values are the same as in the previous example. However, in this case, instead of taking the maximum value of each region, the average value of that region is computed.

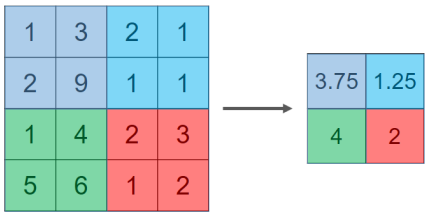


Figure 4-36: Average pooling example with f=2 and s=2

In average pooling, each region of the feature map is downsampled to the average value in the region.

An important thing to note is that these operations, unlike convolutional layers, are performed on 2D matrices. As seen in Figure 4-34, the convolutional operation is applied to all channels at once, and the output volume contains different stacked feature maps, one feature map for each different filter used. So the output volume of the convolutional layer will be the input volume of a pooling layer. Therefore, the pooling layer will input a volume consisting of different stacked 2D matrices (one on each channel) and pooling will be done on each channel independently. It makes sense that if feature maps arrive separately, one on each channel, pooling is done to each of them separately, looking at a single feature at a time. Consequently, the output will have the same number of channels as the input.

The formulas developed in the previous sections for figuring out the output size for convolutional layers also work for pooling layers. Therefore, if the input of a pooling layer is a volume of size:

$$n_H \; x \; n_w \; x \; n_c$$

Then, the layer will output a volume of size:

$$\left\lfloor \frac{n_H - 2p - f}{s} + 1 \right\rfloor \; x \; \left\lfloor \frac{n_w - 2p - f}{s} + 1 \right\rfloor \; x \; n_c$$

Note that the number of input channels is equal to the number of output channels.

The most commonly used values for hyper-parameters are $f = 2$ and $s = 2$. As can be seen in both examples, taking these values has the effect of shrinking the height and width of the representation by a factor of 2. Reducing the input dimensions will reduce the number of parameters that will be used in the next layer, and thus the computation time. For example, using the filter size and the stride of these examples, 75% of the data is discarded.

The other advantage on using pooling is the possibility to make some of the features detected by CNNs more robust. As it was seen in the vertical edge detector example, a limitation of the feature map output of a convolutional layer is that they record the precise position of features in the input. This means that small movements in the position of the feature in the input image will result in a different feature map. This problem can be solved using the downsampling strategy; a lower resolution version of an input is created that still contains the large or important structural elements, without the fine detail that may not be as useful to the task [17]. Therefore, the result is a summarized version of the features detected in the input. They are useful as small changes in the feature location in the input, detected by the convolutional layer, will result in a pooled feature map with the feature in the same location. This capability added by pooling is called the model's invariance to local translation. "Invariance to translation" means that although the input suffers a small translation, values of most of the pooled outputs do not change [17].

In addition, introducing pooling layers helps control overfitting by reducing the size of the input feature map. If the entire feature map were used, there could be overfitting by finding patterns that do not exist, due to the large number of features or the low number of training examples available. Conversely, when selecting a subset of features, false patterns are less likely to be found.

4.4.6.3. Fully connected layer (FC)

Fully connected layers are an essential component of convolutional neural networks. The CNN process begins with convolution and pooling, breaking down the image into features and analysing them independently. The result of this process is fed into a fully connected neural network structure that drives the final classification decision. Fully connected layers have proven to be very successful in image recognition and classification for computer vision [18].

Therefore, the objective of a FC layer is to take the results of the convolution/pooling process and use them, for example, to classify the image into a label if it is an image classification task. To do this, first, the output of the convolution and pooling process is flattened into a single vector of values, each representing a probability that a certain feature belongs to a label. For example, if it is a cat classifier, features representing things like whiskers should have high probabilities for the “cat” label [18]. This vector is then passed through the fully connected layers, and then generally passed through the softmax layer to output the target classification.

A FC layer is like a standard neural network layer seen in the previous sections where each neuron is connected to all activations of the previous layer. Their activations can be computed using a matrix multiplication followed by a bias offset, and then, a non-linear activation function.

$$z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]} \qquad \textit{Eq. 4-72}$$

$$a^{[l]} = g^{[l]}(z^{[l]}) \qquad \textit{Eq. 4-73}$$

Figure 4-37 shows a fully connected layer, which has a weight matrix of dimensions 120x400 and a bias vector of dimensions 120x1. In this case, there are 400 units densely connected to 120 units, and this is fully connected because each of the 400 units is connected to each of the 120 units. Finally, the output passes through the softmax unit where the classification will take place.

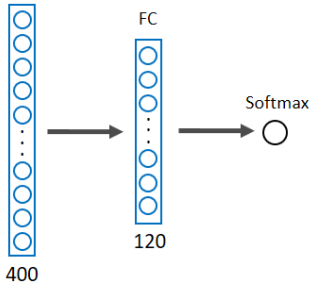


Figure 4-37: Fully Connected layer example

5. Hardware and software equipment

5.1. Neural Network Framework: PyTorch

During these years, the Deep Learning community has been developing different frameworks to help researchers with their projects. Nowadays, there are several different deep learning frameworks, each with their own strengths, weaknesses, and user base.

PyTorch is one of the most popular Open Source Deep Learning frameworks. Both researchers and developers use it because it provides fast and flexible experimentation and a seamless transition to production deployment [19].

PyTorch is a Python-based scientific computing package with increasing popularity among the Deep Learning community. This framework is built to harness the power of GPUs for faster training and, since it is deeply integrated into Python, a very popular language in the research community, it makes it easy to get started [19].

PyTorch provides two high-level features [19]:

- Tensor computation (like NumPy) with strong GPU acceleration
- Deep neural networks built on a tape-based autograd system

Two new concepts are introduced to explain the utility of PyTorch.

PyTorch Tensors [19]:

Tensors are similar to NumPy's arrays, with the added advantage that Tensors can also be used on a GPU to speed up computing. For modern deep neural networks, GPUs often provide accelerations of 50x or greater.

PyTorch provides many functions for operating on Tensors to accelerate the computation and fit the scientific computation needs, such as slicing, indexing, math operations, reductions, etc.

Autograd [19]:

The core to all neural networks in PyTorch is the autograd package, which provides automatic differentiation for all operations on Tensors.

Manual implementation of the backward pass is not a big deal for a 2-layer network, but it can quickly become very complex for large networks. Fortunately, in PyTorch, automatic differentiation can be used to automate the computation of backward passes in NNs.

This makes implementation easier, as the programmer only needs to define the NN with its learnable parameters (weights and biases) and the forward pass, and the backward pass is set automatically. In PyTorch, weights and biases are just regular tensors with a special

addition: we tell PyTorch that they require a gradient. This causes PyTorch to record all the operations performed on the Tensors, so that it can calculate the gradient during back-propagation automatically.

Furthermore, PyTorch extends a rich ecosystem of tools and libraries and supports development in computer vision [19]. PyTorch provides the designed modules and classes torch.nn, torch.optim, Dataset, and DataLoader to help create and train neural networks.

In conclusion, PyTorch is used either as a replacement for NumPy to use the power of GPUs or as a deep learning research platform that provides maximum flexibility and speed.

Furthermore, in this project, the PyTorch framework is used because it is recommended to be used for small projects and prototyping for research purposes [20] and because it is the common framework used in Institut de Robòtica I Informàtica (IRI). The reason is that, thanks to all these libraries it offers, the training process is simple and clear, making it easy to get started if you are learning how to implement a Deep Learning algorithm. Also, using this framework can really speed up the implementation of NNs and avoid making mistakes like those that might appear if the code were implemented from scratch.

Figure 5-1: PyTorch logo [19]

5.2. A visualization tool: TensorBoard

TensorBoard is a tool designed to visualize the results of neural network training runs [19].

Deep learning models are often complex, hard to understand and used as a “black box”. Making the model more accessible and visualizing its progress can help you better understand what is happening during the training process. TensorBoard is the tool needed to inspect what’s inside this “black box” to help catch errors early, debug more effectively and eliminate any bias in the dataset [21].

In machine learning, to improve something, you often need to be able to measure it. TensorBoard is a tool that provides the measurements and visualizations needed during the machine learning workflow to understand and optimize the model. It enables tracking and visualizing metrics (scalar values) such as loss and accuracy, visualizing the model graph, viewing histograms of weights, biases or other tensors as they change over time, displaying images, text and audio data, and much more [22].

This interface is also very useful for comparing different training runs. For example, if a model has been trained and then some hyper-parameters have been adjusted and the model has

been trained again, both runs can be displayed at TensorBoard simultaneously to indicate possible differences [23].

TensorBoard is the TensorFlow's visualization toolkit. TensorFlow, like PyTorch, is a popular framework for Deep Learning. Although TensorBoard is a visualization library for TensorFlow, it can also be used for PyTorch, as it has recently been officially supported by PyTorch [19].

Once TensorBoard is installed, it allows you to log PyTorch models and metrics into a directory for visualization within the TensorBoard UI. The SummaryWriter class is the main entry to log data in .event files [19]. TensorBoard then runs as a (Python-based) web server from the command line, reads the event files generated by external code (like TensorFlow or PyTorch code) and visualizes them in the browser [24].

As mentioned, TensorBoard provides basic functionalities like plotting scalar variables, images, graphs or histograms. In this project, two of all of these will be used:

- Plotting scalar variables: the loss, the accuracy and the learning rate
- Plotting images

Perhaps the most intuitive way to examine the performance of the model is by following the scalar metrics outputs of loss or accuracy. Machine learning involves understanding these metrics and how they change as training progresses. These metrics can help understand if there is overfit, for example, or if the train is unnecessarily too long. Furthermore, they will also be useful for comparing these metrics across different training runs to help debug and improve the model [22].

On the other hand, the images to be plotted are the network output activation maps superimposed on the input images. An interesting objective of image logging might be to track the learning progression of the model as it runs. For example, in this object detection deep learning project, it will be useful to visualize the predicted keypoint over time to see if the algorithm is learning where to place it.

5.3. ROS (Robot Operating System)

ROS is an open source robot operating system.

The Robot Operating System provides a structured communications layer for writing robot software. It is a collection of tools and libraries that aim to simplify the developer's task of creating complex and robust robot behaviour across a wide variety of robotic platforms [25].

It provides services such as hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. And it also provides tools and libraries for obtaining, building, writing, and running code across multiple computers [26].

The concept of “hardware abstraction” comes from the idea that when ROS is used, it is possible to abstract the hardware from the software. This implies creating programs for robots thinking in terms of software for all the hardware of the robot. In this way, it is possible to create programs simply by knowing the robots ROS API, without having to deal with hardware [27].

The ROS API is the list of ROS topics, services, action servers, and messages that a robot provides to give access to its hardware (sensors and actuators). It will be explained later, but the topics, services and messages are like the software functions that the robot can call to get data from the sensors or to cause the actuators perform some action [27].

ROS was built from the ground up to encourage collaborative robotics software development. In this way, different laboratories or groups of researchers could collaborate and build upon each other's work, sharing information and adding knowledge. Over the past several years, ROS has grown to include a large community of users worldwide, and today ROS is one of the most developed and maintained robotics framework for robot programmers [25].

Additionally, ROS was designed to be as distributed and modular as possible, so that users can choose to use packages contributed by others that may be useful for their project. In other words, it seeks to be able to reuse parts that are already programmed without having to program them from scratch [25].

Robot Operating System, despite its name, is not a real operating system. ROS is a middleware, a low-level “framework” based on an existing operating system. The main supported operating system for ROS is Ubuntu. So, ROS is mainly composed of [28]:

- A core (middleware) with communication tools
- A set of plug and play libraries

CORE (MIDDLEWARE) WITH COMMUNICATION TOOLS

At the lowest level, a middleware is a message passing interface that provides communication between processes. When developing new robotics software, it is important to be able to create subprograms (modules), one for each functionality of the application, in order to develop and run it without the whole application [28]. All these subprograms need to communicate with each other, and this is solved with the ROS computation graph.

A computation graph is a peer-to-peer network of ROS processes (potentially distributed across machines) that process data together using the ROS communication infrastructure. ROS implements several different styles of communication, including synchronous communication, asynchronous streaming of data, and data storage [26].

With ROS, the code can be easily separated into packages containing small programs, called nodes [28]. So packages are the main unit for organizing software in ROS and nodes are executable files that perform computation. For example, there will be a node that controls

the wheel motors, another node to perform localization, another to perform path planning and so on [26].

These programs (nodes) communicate with each other through the following three types of ROS communications:

- Topics: asynchronous communication between publishers and subscribers. A node (*publisher*) sends a message by publishing it to a given topic. The topic is a name used to identify the content of the message. Then, another node (*subscriber*) subscribes to this topic if it is interested in the content of the message. There may be more than one publisher and subscriber for a single topic [26].
- Services: synchronous communication between a client and a server (one-to-one communication). This type of communication is useful for request and reply interactions such as changing a setting on the robot or requesting a specific action. One node (server) provides a service under a name and another node (client) uses the service by sending the request message and waiting for the reply [26].
- Actions: asynchronous communication between a client and a server. It is based on topics and there is feedback.

LIBRARIES

In addition to the core middleware components, ROS provides common robot-specific libraries and tools that will make work easier. For example, ROS provides: mapping, navigation, robot geometry library, localization and pose estimation [25].

The main ROS goal is to support code *reuse* in robotics research and development. As indicated previously, there are a large number of open source packages that can be shared, distributed and used independently in the robot used [26].

There are other important advantages when using ROS. For example, the programming language. When writing code, people have preferences for a programming language. Fortunately, ROS is language agnostic, meaning that the programs can be written in any language. ROS is compatible with C++, Python, Octave and LISP languages [29].

In this project, the programs will be written in Python and there will be two nodes, the ZED node to control the camera and set its parameters and the node with all the deep learning computations. These two nodes are going to communicate with each other through topics.

Figure 5-2: ROS logo [26]

5.4. Jetson AGX Xavier

NVIDIA Jetson is the world's leading embedded AI (Artificial Intelligence) computing platform. NVIDIA Jetson systems provide the performance and energy efficiency necessary to run software on autonomous machines faster and with less power [30].

The Jetson platform includes small form-factor Jetson modules with GPU-accelerated parallel processing, the JetPack SDK with developer tools and comprehensive libraries for building AI applications, along with an ecosystem of partners with services and products that accelerate development [30].

The NVIDIA Jetson AGX Xavier Developer Kit (Figure 5-3) is the latest addition to the Jetson platform.

Figure 5-3: Jetson AGX Xavier developer kit [30]

The two most important parts of this Dev Kit are the Jetson module itself with a thermal solution pre-attached and the carrier board that it connects to [30].

Figure 5-4 shows the Jetson module, a complete AI computer for autonomous machines. It delivers the performance of a GPU workstation in an embedded module under 30W and is capable of more than 30 trillion operations per second for deep learning computer vision tasks. It can operate at 10W, 15W or 30W and delivers more than 10 times energy efficiency and more than 20 times the performance of its predecessor NVIDIA Jetson TX2 [30].

Figure 5-4: Jetson module for the Jetson AGX Xavier developer kit [30]

NVIDIA Jetson AGX Xavier marks a new milestone in computational density, energy efficiency, and AI inference functions in small form-factor devices [31].

SMALL SIZE

At only 100 x 87mm in size Jetson AGX Xavier offers great performance, making it the perfect choice for autonomous machines such as logistic and delivery robots, factory systems and large drones for industry [31].

HIGH-PERFORMANCE

As the world's first computer designed especially for autonomous machines, Jetson AGX Xavier's high-performance can handle algorithms for visual odometry, digital location and mapping, obstacle detection, and decisive route planning for next-generation robots [31].

POWER:

Users can configure operating modes to 10W, 15W, and 30W as needed for their applications, enabling new levels of compute density, energy efficiency, and front-line AI inference functions [31].

Therefore, this development kit exploits all the benefits of the module in terms of power and performance and also provides many different connectors (USB 3.0 Type-A to Type-C, HDMI, SD card, Ethernet among others) while maintaining the small form-factor of the device [30].

5.5. ZED camera

The ZED camera is a 3D Vision Sensor from Stereolabs. Stereolabs is a leading provider of depth and motion sensing technology based on stereo vision. In addition to viewing images, stereo vision also allows depth perception, thus providing a 3D perception of the world [32].

With its two high-resolution cameras, the ZED camera reproduces the way human vision works. Using its two “eyes” and through triangulation, ZED provides a three-dimensional understanding of the scene it observes, allowing any application to become space and motion aware. To do this, both cameras simultaneously transmit the captured images to an external computer device to create a three-dimensional map by comparing the displacement of pixels between the left and right images [33].

This camera has a compact structure and a relatively small size (Figure 5-5), which makes its incorporation into robotic systems or drones simple [34].

Figure 5-5: ZED Stereo Camera composed of stereo 2K cameras with dual 4MP RGB sensors [32]

As mentioned, the ZED captures two synchronized left and right videos of a scene and outputs a full resolution side-by-side color video (Figure 5-6) [33].

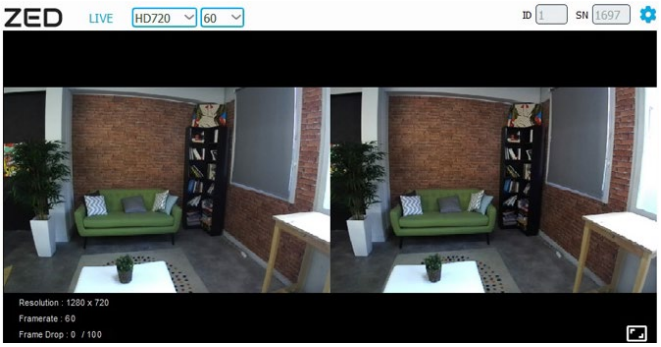


Figure 5-6: Left and right RGB images [33]

This color video is used by the ZED software on the external computer device to create a depth map of the scene, track the camera position and build a 3D map of the area (Figure 5.7) [33].

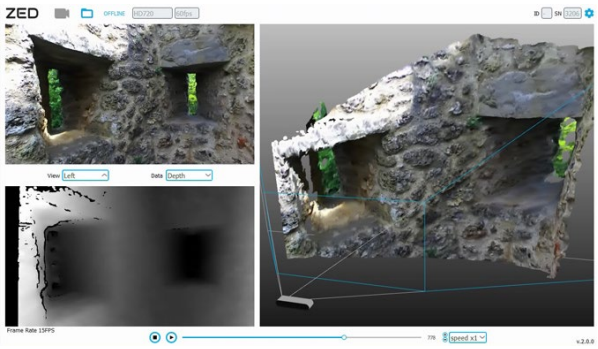


Figure 5-7: Left RGB image, depth map and 3D point cloud [33].

According to manufacturer's information, it is designed and built to perceive the depth of objects in indoor and outdoor environments within a range of 1 to 20 meters and at 100 FPS [34]. This, together with 3D motion tracking capabilities, helps to understand the surroundings in 3D up to 20m distance, which can be useful for applications in many industries: AR/VR, drones, robotics, retail, visual effects and more [33].

This camera has a wide field of view of 90° (H) x 60° (V) x 100° (D) and a high frame-rate, as it can capture 1080p HD video at 30 FPS or WVGA at 100 FPS [33].

Several configurations parameters such as resolution, brightness, contrast, saturation can be adjusted through the ZED SDK (Software Development Kit) provided by Stereolabs. The ZED SDK is available for Windows, Linux and NVIDIA Jetson platforms and allows you to use optimized functions to obtain the maximum performance of the device [34].

In this project, the ZED camera will be connected to the Jetson AGX Xavier and, using the ZED SDK (and ROS), it will be possible to acquire the rectified RGB images for the deep learning algorithm and the depth map to extract the distance value.

6. Object detection

6.1. Deep Learning

6.1.1. Object detection task

The objective of this project is to locate a person in front of the robot combining deep learning and stereo vision techniques (Figure 6-1).

In short, the following will be done: the stereo camera will send the color image and the depth map to the Jetson AGX Xavier. There, the deep learning algorithm is going to carry out the person detection and from its 2D coordinates, the distance value will be extracted to locate the person in the 3D world.

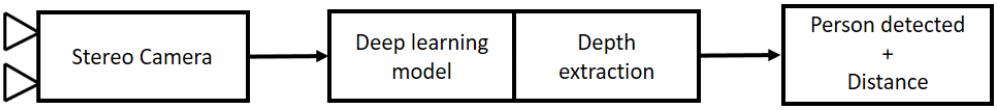


Figure 6-1: Combining deep learning and stereo vision techniques

To locate the person in the 3D world, the point coordinates between their eyes will be used. So, the deep learning algorithm will recognize this point and send its coordinates (x, y) to the distance extraction algorithm. This point will be called the keypoint.

This point has been chosen because it is located in a characteristic region of the face; for example, the eyes are around the keypoint. By having these specific characteristics, it will be easy for the algorithm to learn this point and the region around it.

In addition, although it seems obvious, the region will have similar characteristics in any person, and it will be certain that this region is always going to be found in the same place (on people's faces). Therefore, there should be no confusion problems. It means that the algorithm likeliest detects the point where it should be located, or the keypoint is not going to be detected. Detecting it elsewhere is highly unlikely.

As result of this decision, as the point is between the eyes, if the algorithm detects this keypoint, this means that the person in front the camera is facing.

In this case, the neural network is going to learn the region around this keypoint. To do that, a heat map is created using a Gaussian function. This Gaussian will be centred on the keypoint and the shape will depend on the sigma value. The keypoint in the Gaussian function will have the maximum value (1), and then the values will decrease as the points (pixels) are moving away from the keypoint. Thus, the labeled heat map is a probabilistic function (from 0 to 1).

Therefore, the NN will output a predicted heat map, where the value in each pixel is going to represent the probability that the keypoint is located in that pixel. Then, from the pixel with the

highest probability, a threshold value will be used to select whether that probability is sufficient to say that there is a person in the image; it would mean that specifically pixel corresponds to the point between their eyes. Finally, the coordinates of this point (keypoint) will be used to obtain its depth coordinate using the stereo vision.

To help the network learn this region, many images are collected to build a large enough dataset. This dataset will be used to train and validate the model. In fact, there will be 2748 training images and 306 test images. Before entering the neural network, these images must be reduced and cropped to the network size.

To train the network, the PyTorch framework is used. To validate the model, some evaluation metrics must also be defined, such as accuracy and the confusion matrix; two typical metrics used in deep learning.

The network used has a U-Net architecture, consisting of 14 convolutional layers, the same as that explained in section 2. This architecture will be taken from an old project by an IRI member and the deep learning model will be adapted to this specific implementation.

At last, the images pass through U-Net and the predicted heat map will be obtained. Then, using a loss function, this heat map will be compared to the ground truth heat map (the true output). Finally, with the backward pass, the derivatives will be calculated and the learnable parameters will be updated.

Once the model is trained, it can be implemented in real-time systems.

6.1.2. Dataset

This section explains how the dataset is built. To do that, is necessary to explain what types of datasets are used and how the images are divided among them. Then, is necessary to explain how the dataset is prepared; where the images are taken from, how it is resized, how it is labeled and how the labels are extracted in order to use them.

6.1.2.1. Training set, development set and test set

In deep learning algorithms, data can be divided into three different sets: the training set, the development set, and the test set. Making good choices about how to set up these datasets can make a big difference in helping quickly to find a high-performance neural network.

It is important to make decisions that can help go through the iterative process (idea-code-experiment) efficiently in order to test different sets of hyper-parameters and find a good network for the specific application. And setting up these previous datasets well can make it more efficient.

The data can be splitted up into three different sets:

- **Training set**: this set is used to train the models (weights and biases). A model *sees* and *learns* from this data [35].
- **Development set**: this set, which can also be called the hold-out cross validation set, is used to see which of the many different models performs best on this set. This data is used to evaluate a given model and fine-tune the model hyper-parameters according to the results obtained. Hence the model occasionally *sees* this data but never *learns* from it [35].
- **Test set**: once the final tuned model is chosen, this set is used to evaluate the model in order to get an unbiased estimate of how well the algorithm is doing. The test set is only used once the model is completely trained, i.e. after the training and development sets have been used. This set might be useful when comparing different final models, for example, in a competition between other programmers, or when comparing two different techniques for face detection.

Typically, a larger portion of the data is for the training set, while the development and test sets contain a much smaller percentage of the total data (Figure 6-2). This is because the development set should only be large enough to evaluate different algorithm options and quickly decide which one works best. And the same applies to the test set, which should only be large enough to evaluate a single model.

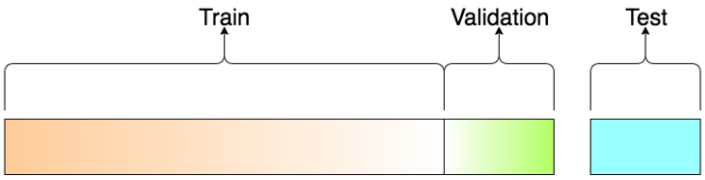


Figure 6-2: A visualization of data split [35].

For some applications, researchers do not have a test set. The purpose of the test set is to provide an unbiased estimate of the performance of the selected model. If this estimate is not required, then the test set is not necessary. And this is the case of this project, since the important thing will be to have some data to train the algorithm, and have some data to tune the hyper-parameters, and thus be able to apply the final model to see how it works.

Therefore, the model must be trained on the training set and evaluated on the development set. Then the results on the development set are used to iterate and find a good model. In this project, the development set is called validation set or test set, interchangeably. Many authors, if they don't have a test set, they call the development set as “test set”.

When setting up a machine learning problem, if the dataset contained a million training examples, it would be more than enough to have 10000 examples in the development set to evaluate which algorithms work best. So in that case, it might be possible to have 99% of the data for training and 1% for evaluation. On the other hand, when the dataset is smaller (with

10000, 1000 or even 100 samples), what is generally done is to set a larger portion of the data for the development set, this being 10%, 20% or even 30% of the data.

In this project, the amount of data collected provides a dataset with fewer than 10000 samples. For this reason, it has been decided to have 90% of the data in the training set and 10% in the validation set. One important thing to note is that this split must be the same during the analysis phase to be able to compare different models under the same conditions. Therefore, this split will be done at the beginning and will remain interchangeable during the different training runs.

However, this distribution is generally based on the own experience, so it requires have carried out different experiments to know how to choose the correct distribution. As this is the first deep learning project carried out, it has been decided to test this 90-10% split first and, if the results were not as expected, the distribution could be changed and in addition all the experiments would have to be repeated again. However, as will be seen later, it will not be necessary to modify this distribution, since the development set will be large enough to accurately represent the performance of the model.

6.1.2.2. Dataset preparation

The dataset preparation is a very important part of the project. Choosing a good dataset can lead to success; and choosing the wrong dataset can be a huge waste of time. Preparing the dataset is a complex and time-consuming task. It has different steps.

1. **Data collection:**

Deep learning projects tend to be long, with many months and years of research and experimentation. However, in this case there is not so much time. In order to spend more time developing the model and not spending too much time on data collection, it is recommended to search for already well-defined datasets and not create one from scratch [36]. In this project, a new dataset has been built using pre-existing datasets.

There are many free public data sets on the Internet that can be downloaded and used in machine learning projects. In this project, in order to detect people's faces, datasets of faces and human bodies have been used. To obtain a more varied and generalized dataset, different datasets have been combined:

- Labeled Faces in the Wild (LFW): LFW is a public face image dataset for face verification. The dataset contains more than 13K faces images collected [37].
- Celebrities in Frontal-Profile in the Wild (CFPW): CFPW is a public dataset of images of celebrities in frontal and profile views [38]. This dataset contains images of 500 individuals. These individuals are politicians, athletes and entertainers, chosen trying to have as many men as women and trying to maintain as much racial diversity as possible.

- Common Objects in Context (COCO): COCO is a large-scale object detection, image segmentation and image captioning dataset. This public dataset contains 330K images with more than 80 object categories [39]. About half of these images contain people.
- Human Co-Detection Dataset (HCD): HCD is a public dataset which contains around 400 images of several persons in different poses and clothes [40].
- VGG Human Pose Estimation Datasets: This is a set of large video datasets with human upper-body pose. It contains the YouTube Pose and the Short BBC Pose datasets. The YouTube Pose dataset is a collection of 50 videos found on YouTube covering a broad range of activities and people. Short BBC Pose dataset contains five one-hour-long videos with sign language signers [41].
- Barcelona Dataset: This public dataset contains 15150 images of urban views of different cities, offices, cars, etc. Among these images, 279 images are of the street scenes from Barcelona [42]. The images can also contain pedestrians.
- INRIA Person Dataset: This dataset was collected as a part of research work on detection of upright people in images and videos. It contains images of standing and walking people [43].

First, different images have been selected from these datasets and separated into positive and negative samples. The deep learning algorithm will learn from these two types of samples. This is why the selection of images is very important to succeed in training a neural network.

Positive samples:

As previously stated, the objective of this project is to detect the point that is between the person's eyes. Positive samples are images that contain this desired point and therefore are images of people in front view.

Hence, from the previous datasets, different images of people in front view have been selected. For example, from the first two datasets, different faces images have been selected (Figure 6-3).

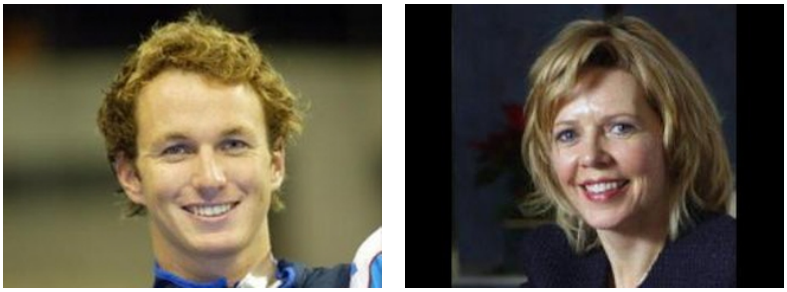


Figure 6-3: Examples of positive samples. Images of human faces in front view [37] [38]

Then, from the COCO, HCD and VGG datasets, images of people doing different activities and in different everyday situations have been selected (Figure 6-4).

Figure 6-4: Examples of positive samples. Images of people doing activities in everyday situations [39] [40] [41]

And finally, from the last two datasets, images of people standing, walking the streets or working in offices have been selected (Figure 6-5).

Figure 6-5: Examples of positives samples. Images of people working in offices and walking the streets [42] [43]

Finally, 3054 positive samples have been selected. About half of these are faces images (from the first two datasets, [37] and [38]) and the other half are images from the rest of the datasets, where there are people near and far from the camera.

Another key to success in neural network training is having a balanced and diverse data set.

In real life, images can be different in terms of lighting conditions, scaling factors, rotation angles, image resolution, and especially people images may be different due to the person itself, and the position of his body and his face. Therefore, it is important to have different types of data to help the network better understand what it has to learn and the different types of data that it may find in the future. This will allow the network to locate people even if conditions are not ideal.

In this project, the following factors have been taken into account:

- Different illumination conditions

Figure 6-6: Examples of positive samples. Images under different illumination conditions [39] [41]

- Different resolution

Figure 6-7: Examples of positive samples. Images of different resolution [39]

- Blurred images

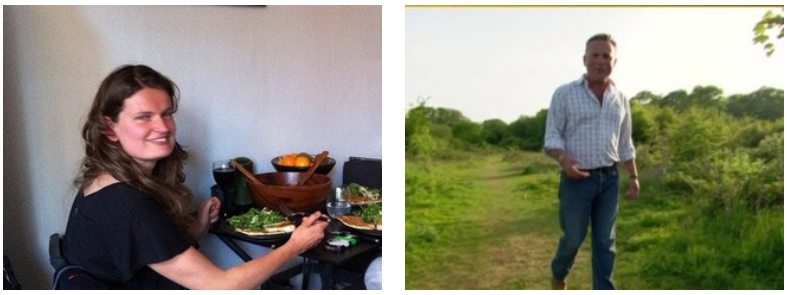


Figure 6-8: Examples of positive samples. Blurred images [39] [41]

- Different perspective

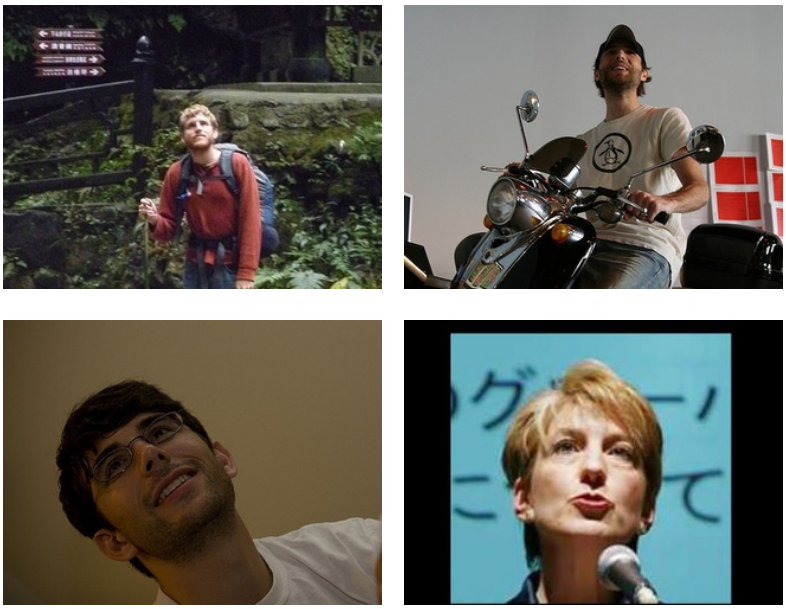


Figure 6-9: Examples of positive samples. Images of people viewed from below [37] [39]

- People with different head orientation

Figure 6-10: Examples of positive samples. Images of people with different head orientation; above, orientation of the head in terms of roll movement and below, orientation of the head in terms of yaw movement [37] [38] [39].

- People at different distances

Figure 6-11: Examples of positive samples. Images of people at a greater distance [39] [42]

- People of different genders and ages

Figure 6-12: Examples of positive samples. Images of people of different genders and ages [39] [41]

- People from different countries

Figure 6-13: Examples of positive samples. Images of people from different countries [39]

- People with glasses

Figure 6-14: Examples of positive samples. Images with people with glasses [39]

- People with different facial expressions

Figure 6-15: Examples of positive samples. Images of people with different facial expressions [39]

- People with slightly closed eyes

Figure 6-16: Examples of positives samples. Images of people with slightly closed eyes [37] [41]

Considering that the neural network is used to locate people and will be implemented in a robot for multiple applications (such as approaching the person to deliver a package), these factors are important for the following reasons.

For example, in case the camera is below people's faces, it is important to have images of people viewed from below (Figure 6-9). In addition, it is also important to have images with different perspectives and head orientations in case the camera rotates, or in case the person turns his head at any time, or in case the robot approaches the person from another angle, that is, that it does not approach exactly from the front.

It is also important to detect people with their eyes slightly closed, because in addition to some people who already have their eyes slightly closed, we close our eyes many times; it is important not to lose the location of the person in these cases.

Also, these people can be outside or inside a building, where there may be shadows or brighter areas, so it is important to have images with different illumination conditions. In addition, the resolution of the images changes in each camera, so it is important to have images with different resolutions.

Furthermore, it is important to have images of people who wear glasses because glasses will affect the region that the network learns. For this reason, the network is also required to learn that there may also be glasses in this region. Images of people with different facial expressions

are also taken for the same reason, as expressions can change the characteristics of the learning area, for example by adding expression wrinkles.

Thus, when the model is implemented in a robot, this robot will be able to locate people of different ages and gender, moving or static, at different distances, even in adverse conditions. Then, the robot will approach the person knowing where it is to avoid colliding.

Negative samples:

Negative samples are images of people in profile view, people from behind or images without people. It is important that the point between the eyes does not appear in this images.

These images have also been obtained from previous datasets. For example, from the first two datasets, different images of human faces in profile view have been selected (Figure 6-17).

Figure 6-17: Examples of negatives samples [37] [38]

Then, from the other datasets, images of different people in different situations and also images without people have been selected. Among these images are images of the streets of Barcelona, offices and landscapes (Figure 6-18).

a)

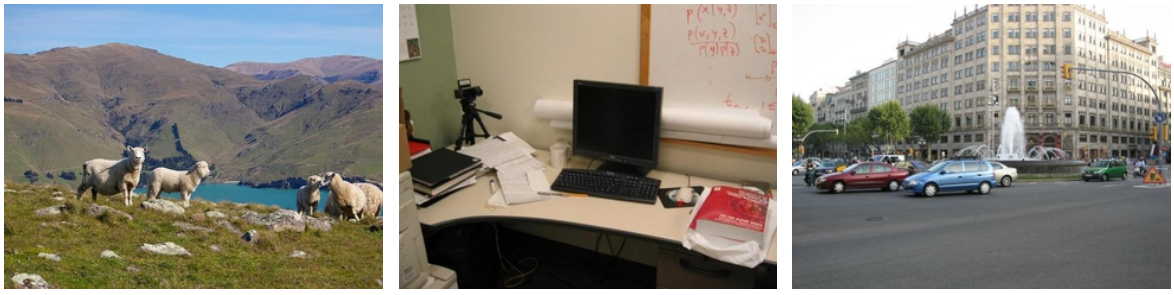
b)

c)

d)

e)

f)



g) h) i)

Figure 6-18: Examples of negative samples. Images of a) and b) people from behind, c) and d) people in profile, e) and f) people walking the streets, g) a landscape, h) an office and i) the streets of Barcelona [39] [41] [41] [42] [43].

Negative samples are just as important as positive samples, since the algorithm can also learn from them that there are no people to locate there.

Finally, around 6000 negative samples have been selected.

It is important to note that images of the same person have been taken as positive and negative samples, as well as images of people doing the same activities (such as playing tennis or working) and images with the same backgrounds (such as tennis courts, offices, streets) (Figure 6-19). This prevents the network from learning unwanted patterns.

Figure 6-19: Samples with the same person. The negative image on the left and the positive, on the right [41].

2. **Dataset resizing:**

Until now, the dataset contained images of different sizes and different aspect ratios. This section explains some decisions that have been made to avoid possible errors in image preprocessing and to speed up image processing as they pass through the neural network.

At this point, it is necessary to first explain what decision has been made regarding the size of the network input images. It has been decided that the U-Net will be trained with relatively small square images, specifically images of sizes 224x224 and 180x180. These dimensions have been chosen because they are standard sizes of square images used to train NNs.

Furthermore, it has been decided to train the network with relatively small images for two reasons:

First, because it is important a fast image processing in order to locate people in real time. A small image, since it contains less data, will be able to pass through the network faster than a larger image.

And second, because most of the images in the dataset are small. By choosing to train the network with relatively small images, this avoids having to upscale all these small images. Upscaling, unlike downscaling, would adversely affect image quality, causing a loss of focus when trying to fill in the image with missing information. Therefore, if a small image size is chosen, small images will not have to be upscaled and their quality will not be affected, and images that are larger can be downscaled without affecting their quality.

Before cropping the images to the network size, it has been decided to resize all the images in the dataset to the same size, which will be 320x240, also a relatively small size. The decision to resize images has been made for two reasons:

Small images: First, because there are many images with dimensions slightly smaller than the dimensions of the network input images, especially in the first two datasets (the face datasets [37] and [38]). As they are small images, they do not fit in the network. For this reason, these images should be slightly larger. The method used to avoid affecting the quality of the images will be explained below.

Large images: On the other hand, there are some images in the dataset that are much larger than the network input images. So here two situations can occur:

- If the image contains a close-up person, when cropping the image to enter the network, some important information about the region to be learned is lost (such as the eyes, as can be seen in Figure 6-20).
- If the image contains a person at a greater distance, when cropping the image, the sense of distance is lost (Figure 6-20). In this project, this will be decisive.

Also, trying to reduce the image size directly to the size of the network input is not an option, since, as these are images with a different aspect ratio, the image can be deformed and with it, the person and the region to be learned.

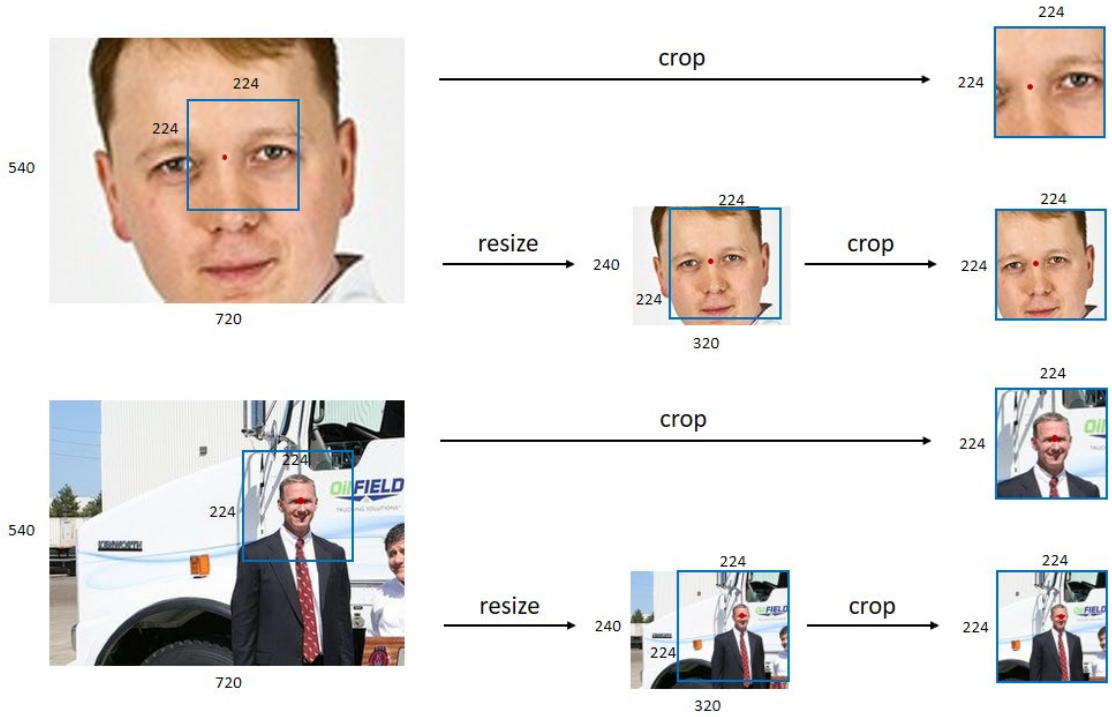


Figure 6-20: Problems and solutions for large images. Above, there is a close-up image of a person. When the image is cropped, the relevant information is lost. If it is resized first, the image will contain the relevant information. Below, there is a person at a greater distance. When the image is cropped, the sense of distance is lost. If it is resized first, the sense of distance remains, since the person seems to be still at a greater distance.

Image resizing will ensure that after cropping the images to the network image size, they still contain the desired region.

To resize the dataset, different techniques have been used for small and large images.

Images that are smaller than 320x240:

In this case, since upscaling is not good, the padding technique is used. This consists of adding contours with pixels of intensity 0 to the image. As can be seen in Figure 6-21, black borders are added to the image until having the desired dimensions.

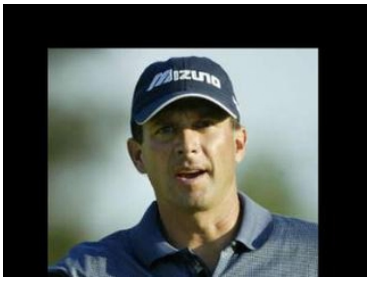


Figure 6-21: Example of image padding

In these cases, it is recommended to add the minimum border to the image, since an image with more padding than the image itself would not make sense. Also, the network has to learn

that black borders are part of the padding operation and not part of the image. For this reason, small images are padded only until they have the desired dimensions (320x240).

Images that are larger than 320x240:

For images larger than 320x240, cropping is the technique used.

This preprocessing technique has been done taking into account the smallest dimension of the image, the width or the height (Figure 6-22), and from there, the image has been cropped with the same aspect ratio as 320x240, that is, 1 1/3. This is done so that when the image is resized to 320x240, the aspect ratio is preserved and the image is not deformed.

Figure 6-22: Examples of image cropping

Once all these images, both positive and negative, have the same aspect ratio, they are resized to 320x240.

Finally, using padding for small images and combining cropping with resizing for large images, it is possible to have all the images in the dataset with 320x240 dimensions. From these images, the network input square images can be obtained using the cropping technique.

The purpose of resizing all images to a size of 320x240 is to produce a smaller dataset size. Just as small images speed up image processing (within the NN), having a dataset with small images also speeds up image preprocessing (before entering the NN). Image preprocessing consists of a series of operations that, since they can operate with small images and images of the same size, can be programmed more easily and performed more quickly.

The specific image size of 320x240 has been chosen for two reasons:

Firstly, because it is a fairly typical image size and many images already had these dimensions. And second, because the size of 320x240 works well to be able to crop the images to the size of the network without losing much information and, at the same time, leave a margin to be able to crop the images at random in each epoch. This technique will be explained later.

3. Dataset labeling:

The next step is to label the positive samples. This operation consists of creating, for each positive image, a file with the annotations of the keypoint location. More specifically, the keypoint coordinates are extracted and stored in a XML (Extensible Markup Language) file.

As a remainder, the label of a sample $x^{(i)}$ is the final output $y^{(i)}$ that the model tries to predict ($\hat{y}^{(i)}$). In this case this is a little different; the label (*hm*) will have the 2D coordinates of the point between the eyes of the person in the image. From these coordinates, the heat map is built ($y^{(i)}$) and the NN will try to predict this heat map ($\hat{y}^{(i)}$).

To label the dataset, there is a graphical image annotation tool called LabelImg. LabelImg is one of the most popular graphical image labeling tools written in Python [44].

This tool is generally used to label bounding boxes of objects in images and, as it is programmed, the XML file stores the upper left corner and the lower right corner of the rectangle. For this reason, when manually labeling the images, the upper left corner of the bounding box is made to coincide with the point between the person's eyes (Figure 6-23).

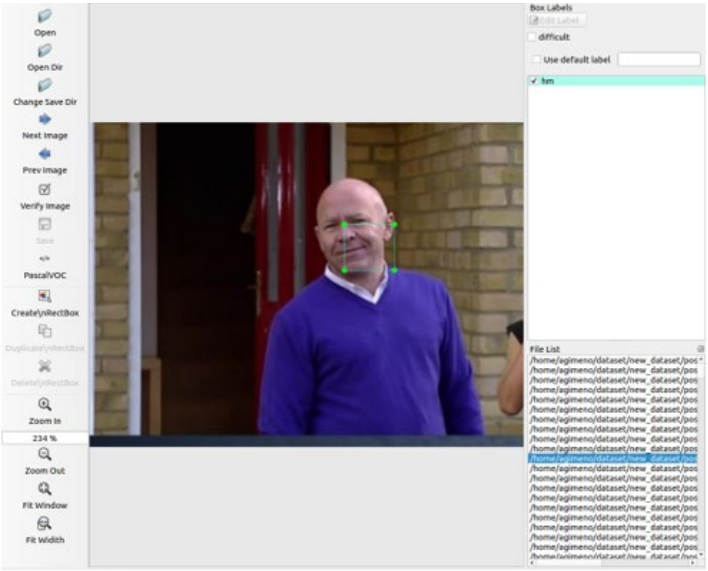


Figure 6-23: Example of image labeling

Then, the XML file (Figure 6-24) will contain the following data:

- Image path: the path to locate the RGB image on the computer.
- Image size: as indicated above, they will be RGB images of size 320x240x3.
- Object: in this project, the label is called “hm” and there is only one for each image.
- Bounding box: the coordinates of the upper left and the lower right corners.

```
<?xml version="1.0"?>
- <annotation>
    <folder>color_resize</folder>
    <filename>240.jpg</filename>
    <path>/home/agimeno/dataset/new_dataset/pos/color_resize/240.jpg</path>
  - <source>
      <database>Unknown</database>
    </source>
  - <size>
      <width>320</width>
      <height>240</height>
      <depth>3</depth>
    </size>
    <segmented>0</segmented>
  - <object>
      <name>hm</name>
      <pose>Unspecified</pose>
      <truncated>0</truncated>
      <difficult>0</difficult>
    - <bndbox>
        <xmin>187</xmin>
        <ymin>75</ymin>
        <xmax>224</xmax>
        <ymax>109</ymax>
      </bndbox>
    </object>
</annotation>
```

Figure 6-24: XML file that corresponds to the labeling of the image in the previous figure

Later, the coordinates of the upper left corner will be used in the model and the others will be discarded. For example, for the image above, the important coordinates are (187,75). It is important to note that the coordinate system is as shown in Figure 6-25.

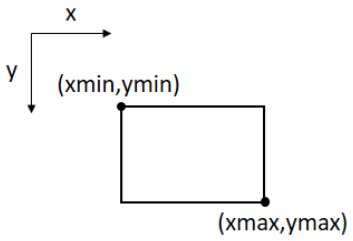


Figure 6-25: Coordinate system in LabelImg and upper left and lower down corner coordinates

Finally, all positive samples are labeled.

4. Extract labels

In this step, the coordinates of the upper left corner are extracted for all images and stored together in another file.

A Python script (Annex 12.2.1) has been used to extract the *hm* labels from all positive samples. Using this script, all the XML files are read and, for each of them, the desired coordinates (xmin, ymin) are extracted and added in a Python dictionary. However, for later reasons, these coordinates are stored in the reversed order (ymin, xmin).

For each positive sample, the sample name and the *hm* label are attached in a Python dictionary:

$$hms = \{'name\_img1': [ymin_1, xmin_1], 'name\_img2': [ymin_2, xmin_2], \dots\}$$

So this Python dictionary will contain the input labels for all the positive samples. For example, for the image in the example above, its data will be added in the Python dictionary as follows:

$$hms = \{\dots, '240': [75, 187], \dots\}$$

where '240' is the name of the image as can be seen in Figure 6-24.

Then, this Python dictionary is stored in a PKL file. A PKL file is a file created by pickle, a Python module that enables objects to be serialized to files on disk and deserialized back into the program at runtime. A PKL file is pickled to save space when being stored or transferred over a network and then is unpickled and loaded back into program memory during runtime [45].

This operation is done because having a PKL file is easier for the model to take the input data. It is faster to extract the data from the Python dictionary instead of having all the data scattered in 3052 different XML files.

5. **Split dataset**

In this section, the dataset is splitted into two parts, the training set and the validation set. As explained previously, 90% of the data is for the training set and 10% is for the validation set.

As a reminder, there are 3052 positive samples and around 6000 negative samples. In this section, the positive samples will be separated into two parts, while the negative samples will be separated later in the training code. This is done to make the next steps easier, since positive images have more information associated with them, such as labels, and they need to be clearly separated.

To do this, a Python script is used (Annex 12.2.2). First, all the positive samples names are stored in a Python list, then this list is randomly shuffled, and finally, the first 90% samples are selected for the training set, and the other 10% samples are for the test set. The names of these selected samples are stored in CSV files; in the "train_ids.csv" file and in the "test_ids.csv" file, respectively.

Since the dataset has been built from different datasets, the samples are randomly shuffled before splitting it into two parts so that each split has the same distribution as the others. For example, if the samples were not shuffled, the training set would contain all the faces images corresponding to the first two datasets.

Finally, the dataset is ready to be used for training the neural network.

6.1.3. Evaluation metrics

When training a model, it is important to have an evaluation metric to see if the model is really being trained and also to be able to compare models with different hyper-parameter values and know which one works best.

Therefore, the evaluation metric will give the information to know what kind of improvements must be done in order to achieve the desired precision. In addition, it will make this iterative improvement process easier and faster.

There are different metrics to evaluate the models. The metric choice will depend on the model and its implementation.

First, since this algorithm outputs a probability value, a threshold value is used to convert this probability to a class value. If the probability value is less than the threshold value, the sample is classified as class 0, and otherwise, it is classified as class 1. In this project, the threshold value determines whether there is a person (class 1) or not (class 0) in the image. Therefore, it becomes a classification problem. From this class value, the different metrics are computed.

When computing the metrics, where the keypoint is found is not taken into account, that is, if the algorithm finds a point with a probability greater than the threshold, it is assumed that there is a person in the image looking at the camera. However, as has been said, it is highly likely that the points with a probability greater than the threshold are points of the learning region. On the other hand, the keypoint location is considered in the cost function.

To better understand the metrics used, the following concepts are introduced:

- True negatives: are the rejections correctly classified as negatives. In this case, the true negatives are the samples that do not contain the keypoint (that is, the negative samples) and are classified as negatives (class 0).
- False positives: are the incorrectly classified as positives. In this case, the false positives are the samples that do not contain the keypoint (a negative sample) but are classified as positives (class 1).
- False Negatives: are the incorrectly classified as negatives. In this case, the false negatives are the samples that contain the keypoint (a positive sample) but are classified as negatives (class 0).
- True positives: are the correctly classified as positives. In this case, the true positives are the samples that contain the keypoint (a positive sample) and are classified as positives (class 1).

During the training and the validation of the model, after each batch, all the samples in the batch are classified using the threshold value into these four different groups. Then, different metrics are computed in order to see their evolution. These operations can be performed at the end of each batch, since they do not require much time.

These values are computed using the training set and the validation set. It is important to know the metric values in both datasets in order to know where the error is. For example, it will be possible to know if there is a large error in the training set (bias error), or there is a low training

error but a higher validation set error (variance error due to overfitting). Then, depending on which case, the improvements will go in one direction or another.

The metrics used in this project are the following:

Accuracy:

The proportion of the total number of predictions that are correct.

This metric is the only one that is computed after each batch using these four groups (TP, TN, FP, FN). This allows you to draw its evolution parallel to the loss function, since the loss function is also computed after each batch. Therefore, the loss function and the accuracy are computed from the predicted output and the labeled input.

With TensorBoard, it will be easy to see its evolution during training to verify if the model is learning or not. If the model is learning, the loss function will decrease to the 0 value and the accuracy will increase to the 100%. If some change is made during training, it will be easy to see if this works better or not than what it previously was.

However, the important value that is useful for comparing different models is the maximum accuracy that is achieved. And this value is obtained in the final epoch, when it is assumed that the parameters have been learned.

The accuracy can be computed as in Eq. 6-1.

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \qquad \textit{Eq. 6-1}$$

Confusion matrix:

A confusion matrix is an NxN matrix where N is the number of classes being predicted. It helps to see the distribution of all the samples in the previous four groups (Figure 6-26).

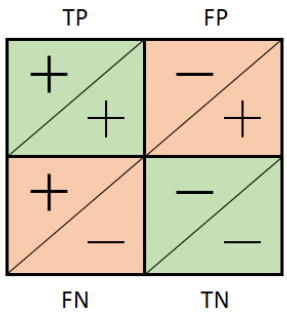


Figure 6-26: Confusion matrix. In each cell, the symbol above the diagonal line is the actual class and the symbol below is the model prediction.

The confusion matrix is computed in the last epoch using the entire training set and the entire validation set. In addition, the samples names that belong to the false positive and the false negative groups, are written to a file in order to see in which type of samples the model fails.

Since in the last epoch the different TP, TN, FP, FN values are obtained, other metrics can be computed to compare the performance of different models.

Precision (P) or positive predicted value:

The proportion of positive cases (predictions) that are correctly defined. In this case, the percentage that the algorithm has to success when it classifies a sample as a positive sample (it determines that there is a person in the image).

Precision can be computed as shown in Eq. 6-2.

$$P = \frac{TP}{TP+FP} \qquad \textit{Eq. 6-2}$$

Negative predicted value:

Similarly, the proportion of negative cases (predictions) that are correctly identified. It can be computed as in Eq. 6-3.

$$Negative\ predicted\ value = \frac{TN}{TN+FN} \qquad \textit{Eq. 6-3}$$

Sensitivity or Recall (R):

The proportion of the actual positive samples that are correctly identified as such. This metric is also known as the true positive rate. In this case, the percentage of actual people (positive samples) that are recognized.

Recall can be computed as shown in Eq. 6-4:

$$R\ (true\ positive\ rate) = \frac{TP}{TP+FN} \qquad \textit{Eq. 6-4}$$

Specificity:

Similarly, the proportion of actual negative samples that are correctly identified as such. This metric is also known as the true negative rate and can be computed as in the Eq. 6-5.

$$True\ negative\ rate = \frac{TN}{TN+FP} \qquad \textit{Eq. 6-5}$$

All of these metrics are useful for understanding the performance of the algorithm. The loss function is useful to see if the model is being trained and is the function to optimize (minimize) during the gradient descent. But it is important to have evaluation metrics in order to obtain specific numbers (proportions) to understand the model performance and to know the direction to make changes in order to improve the model to obtain better results.

6.1.4. Network training

After having the dataset prepared and the evaluation metrics defined, the code available on GitHub [5] written by the IRI's member, is adapted to this dataset and the chosen metrics.

Then this model will be trained using different sets of parameters, hyper-parameter and other training options. Finally, using TensorBoard, the best settings will be chosen to be used with stereo vision.

To train the network with different parameters without having to change all the program code every time, the easiest solution is to modify all the parameters in a single file. When this file is executed, the program runs with all its parameters as command-line arguments (Annex 12.3.2).

This file will be used in this project to change the parameters and hyper-parameters in each training run. Parameters that are not specified here will take the default value if the program needs to use them. The default values are stored in three different scripts (Annex 12.3.1)

These files can include hyper-parameters like learning rate or number of epochs, arguments with information on how to save results, or arguments with information on which GPU to use.

A training epoch represents passing the entire training dataset through the network. Hundreds of epochs are required to train the model and learn the parameters *W* and *b*.

6.1.4.1. Initialization

As a preliminary step to network training, initialization of the dataset, the model and the visualizer is required.

6.1.4.1.1 Dataset initialization

First, the training and the validation iterable sets are created using PyTorch (Annex 12.3.3).

The torch.utils.data.DataLoader class is used to create these two datasets [19]. It represents a Python iterable over a dataset, with support for map-style and iterable-style datasets, customizing data loading order, automatic batching and single and multi-process data loading. These options can be configured by the constructor arguments of a DataLoader:

```
DataLoader(dataset, batch_size=1, shuffle=False, num_workers=0, drop_last=False)
```

All these arguments are explained below.

Dataset:

The most important argument in the DataLoader is "dataset" which indicates a dataset object to load data from.

PyTorch offers two types of datasets, the map-style dataset and the iterable-style dataset. The type of dataset chosen in this project is the map-style dataset which represents a map from keys or indexes to data samples. For example, given a dataset, when accessed with dataset[i], it could read the i-th image and its label from a folder on the disk.

This can be done by defining the __getitem__() method which supports fetching a data sample for a given index. Besides, the __len__() method returns the size of the dataset.

The first step is to read the data from the previous files and the second step is to define the __getitem__() method to generate one sample of data.

As previously mentioned, there are positive and negative samples in the dataset. However, a sample is defined here as the set of a positive sample and a negative sample. Therefore, one sample will contain a positive image with its corresponding label and a negative image.

At this point is important to differentiate how samples are generated for the validation and for the training set.

The positive images are separated from the beginning in the two CSV files, while the negative images are here randomly shuffled and splitted into two parts: the first 10% of the images is for the validation set and 90% is for the training set.

Then, for example, to generate a sample for the validation set, the positive image is chosen randomly among the positive images stored in the test_ids.csv file, and the negative image is chosen randomly from the images in the first part of the list of negative images.

Different ways of generating these datasets have been tried. As will be explained in section 6.1.5, at the beginning, the negative images were not shuffled and the indexes of the positive images were found to repeat it in the same epoch. Then, this is changed by shuffling the negative images and preventing the positive images from repeating in the same epoch. The results of these changes can be seen in section 6.1.5.

After that, common preprocessing techniques to be applied to the images before feeding them into the model are defined.

In the first place, flipping and cropping operations are used in images.

Firstly, one of the four options defined to flip the image is randomly chosen. These options are: flip the image horizontally, vertically, both horizontally and vertically or not flipping the image. Then, the keypoint is modified according to this transformation.

Secondly, the image is cropped to the network image size accordingly to where the keypoint is. Taking into account a minimum margin of separation of the keypoint with the edges of the image, this is cropped randomly to input image size.

By displaying some input images, it has been seen that the margin is not sufficient. The margin is too small, so if the person's face is next to the edge of the image, a significant part of the region around the keypoint (like an eye) is cropped. Therefore, this margin is required to be changed to a higher value. By displaying the input images again, all the positive samples still contain the region to learn (Figure 6-27).

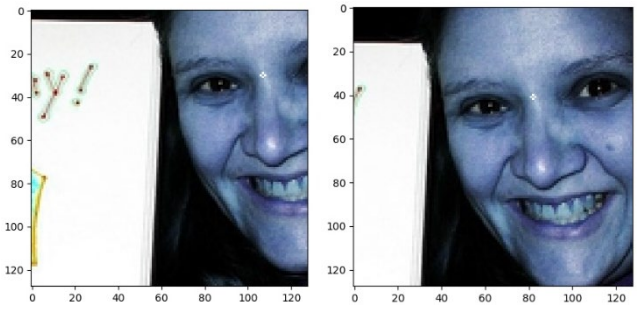


Figure 6-27: Example of image cropping. On the left, with margin=20 and on the right, with margin=40.

Note that this margin will vary depending on the size of the image size that passes through the network. And it is a very important parameter, since if it is too small, a positive sample becomes a negative sample (the image on the left in Figure 6-27). The results of this change can be seen in section 6.1.5.

Since the __getitem__() function is called at each epoch to load the data, these transformations will be performed "on the fly". Therefore, by doing these transformations "on the fly", after each epoch, a new set of randomly transformed samples will be received. This technique is called data augmentation and is used to prevent overfitting.

In the second place, images undergo different transformations using PyTorch's torchvision package. This package contains common image transformations for computer vision. An important transformation is tensor normalization. In this case, the input image tensor to normalize has a shape of C x H x W and its values are already in the range [0.0, 1.0].

To normalize a tensor, instead of calculating the mean and standard deviation of the dataset, what some researchers usually do is use a mean of [0.485, 0.456, 0.406] and a standard deviation of [0.229, 0.224, 0.225]. In this case, for image tensors with values in [0.0, 1.0], the transformation using these values will standardize it, so that the mean of the data will be 0 and the standard deviation 1.

$$\text{Input[channel]} = (\text{input[channel]} - \text{mean[channel]}) / \text{std [channel]}$$

This is done for both positive and negative images. Then, the keypoint coordinates are also normalized using the following equation:

$$hm = \left(\frac{hm}{net\_image\_size - 1} - 0.5\right) * 2$$

It is important to normalize the input data to optimize the algorithm to train faster.

Finally, the __getitem__() method is ready to generate one sample of data and the __len__() method to return the dataset size (i.e. the number of samples). With these methods the dataset object for training and validation are created.

Batch size and drop last:

DataLoader supports automatically collating individual fetched data samples into batches via arguments batch_size and drop_last.

Batches are used to fastest the gradient descent algorithm. This optimization technique consists of splitting up the entire training set into mini-batches. Therefore, a mini-batch will contain N data samples and these samples will be processed at the same time when they are passed through the network. The batch_size argument is used to determine how many samples per batch to load (N).

Without mini-batches, the entire training set (*m* samples) must be passed through the network to perform a single step of the gradient descent method. However, when using mini-batches, parts of the training dataset (N samples) pass through the network, and therefore, the algorithm makes progress before all the *m* examples have been processed.

In this way, a gradient descent step is performed after processing a single batch. The batch size will depend on the GPU used and the network image size. This parameter will be changed so as not to saturate the GPU (section 6.1.5).

The drop_last argument is used to drop the last incomplete batch if the dataset size is not divisible by the batch size. If this argument is not used (set to False) and the size of dataset is not divisible by the batch size, then the last batch will be smaller. Setting drop_last to True avoids problems due to the fact that the last batch is smaller than the rest. Therefore, in this project, it is turned to True.

Shuffle:

For map-style datasets, data loading order is controlled by the shuffle argument. This argument creates sequential batches (if set to False) or shuffled batches (if set to True). In this project, this argument is set to True to have the data reshuffled at each epoch, so the indexes list is randomly permuted to create random mini-batches.

Number of workers:

The mini-batches are loaded at each epoch when the enumerate(dataloader) function is called. The default value for the “number of workers” argument is 0, and that means that the data will be loaded in the main process. Therefore, data loading may block computing.

To avoid blocking computation code with data loading, PyTorch provides an easy switch to perform multi-process data loading by simply setting the argument num_workers to a positive integer. This positive integer determines how many subprocesses to use for data loading. Therefore, after the shuffle randomization is done in the main process, the dataset and the assigned indices to load are passed to each worker, where they are used to initialize and fetch data for mini-batches.

Using more number of workers implies more memory usage, but it fastest the process. The reason is that workers preprocess data batches in parallel when the enumerate(dataloader) is called so that the next batch is ready for use when the current batch has finished.

As can be seen, the dataset initialization is relatively easy using PyTorch. This framework allows you to create iterable datasets, do common image transformations using torchvision, and load random mini-batches into a multi-process data load for training.

6.1.4.1.2 Model initialization

At this step, the object detector model is created (Annex 12.3.4).

NN architecture initialization:

The U-Net architecture explained in section 2, is built using the torch.nn library of PyTorch.

As mentioned in section 2, this network has 14 convolutional layers; five of them are used to increase image size and five to decrease the image size. The script used has not been modified (Annex 12.3.5). This script prepares the layers and the forward step.

At the end of U-Net, a Softmax unit could be used to classify the output into two classes: the keypoint exists in the image (class 1), or not (class 0). In this project, to separate the outputs in these two classes, a threshold value will be used. This threshold value will be used regardless of the gradient descent method and will only be used to calculate evaluation metrics. In this case, we are interested in working with heat maps instead of classes, since it is not a classification task. For example, we want the gradient descent algorithm to optimize the error between heat maps (probability functions), and not the classification error.

The U-Net architecture is moved to the GPU. All deep learning processes are trained there.

Heat map initialization:

As a reminder, the heat map is the CNN output and is what the model is trying to predict. In this step, the heat map is defined for the positive and negative samples in order to prepare the labeled heat map (y) for the training.

In the case of positive samples, the heat map will be a probability function. The maximum probability will be located at the hm coordinates (the keypoint coordinates).

On the other hand, the heat map for the negative samples will be a tensor with all values 0, since there is no keypoint in the image.

Training variables initialization:

As a reminder, the goal of NN is to learn the parameters *W* and *b* using an optimizer (such as the gradient descent method) and the learning rate.

As explained in section 4.1.4, the learning rate value determines the length of the optimizer steps to the optimal point in the cost function. In other words, it determines the speed at which the model learns. This value is an important hyper-parameter and will be changed to see its effect on the training process (section 6.1.5).

In general, the learning rate is slowly reduced over the time (or, what is the same, over the number of epochs). This is another optimization technique frequently used in deep learning and it is called learning rate decay. When the algorithm is far from the target (the minimum of the cost function), the learning process can be faster (the learning steps can be longer), and therefore the learning rate can be higher. But, when the model is closer to the objective, we want the learning process to be slower by reducing the steps length in order to guarantee convergence to the optimal value.

In this project, the learning rate is halved after a certain number of epochs. Therefore, in TensorBoard, the learning rate value is going to be represented as a discrete function. In order to know at which epoch the learning rate should be reduced, it is important to see at which epoch the loss function stabilizes near the value 0; the algorithm has converged. For example, if it converges on epoch number 100, then, the alpha is halved every 100 epochs.

Secondly, the Adam optimizer is defined. In Coursera's online course, it is the most recommended optimizer. This consists of applying two modifications to the gradient descent method. Until the derivatives computation, the method remains the same. Therefore, modifications are introduced in the parameters update. In short, these modifications consist of avoiding the gradient descent oscillations and focusing the direction of the steps towards the optimum point. In this way, the algorithm can learn faster.

Then, this optimizer is defined using the torch.optim.Adam class from PyTorch.

Loss initialization:

For an object detection task, where the output is a heat map, to model the mean of this probability distribution, the mean squared error (MSE) is generally used as the loss function. As previously stated, we want the algorithm to optimize the error between the labeled heat map and the predicted heat map. In this project, the keypoint position is important, not whether

the image contains the point or not. We want the algorithm to learn the point in order to be able to locate the person. And in this case, this loss function is used instead of the one used in a classification task (the one in section 4.1.3).

In PyTorch there is a class in torch.nn to define this function and has the structure [19]:

torch.nn.MSELoss(size_average=None, reduce=None, reduction='mean')

This class creates a criterion that measures the mean squared error between each element in the prediction tensor ($\hat{y}$) and target y. Therefore, the loss will be computed as follows:

$$L(\hat{y}, y) = L = \{l_1, \dots, l_N\}^T \text{ where } l_n = (\hat{y}_n - y_n)^2$$

Where N is the mini-batch size. As the reduction argument is equal to “mean”, then:

$$L = mean(L)$$

Consequently, the loss of a mini-batch is computed by taking the mean of the loss values for each sample.

In this case, there will be a loss function for positive samples and another loss function for negative samples. What is going to be optimized is the total loss, which is the sum of those two. In addition, a loss function will be computed for the training set and for the validation set. Finally, all these loss functions will be visualized at TensorBoard to see if the model is learning.

6.1.4.1.3 Visualizer initialization

The “visualizer” is used to display the results in TensorBoard in order to compare the models.

In this step, the files to write the results to are initialized: a TXT file to save the loss values (Annex 12.3.6) and a JSON file to save all the data for display in TensorBoard.

In this project, the data to visualize will be the different loss functions, the learning rate value, the model accuracy after each batch, and the predicted heat maps.

6.1.4.2. Network training

Once the training and test sets, the model and the visualizer are initialized, the training and the validation processes begin. In this project, the model is trained and validated at the same time. During the training process, every 30s, the results in the current batch are stored to visualize them in TensorBoard. Then, the model validation is carried out using a single random batch and its results are also stored.

A single epoch:

Inside each epoch, the batches are passed one by one through the network, so the training starts with a for loop through all the mini-batches. To load the data of the batch, is here where

the enumerate function is called and the different workers prepare the data of the different mini-batches. As explained above, when loading the data, the __getitem__() function performs random operations “on the fly”. So instead of having the same items at every epoch, random variants are obtained. So, for example, after three epochs, you would have seen three random variants of each sample in the dataset.

For each mini-batch, the steps are the following:

Model training:

1. Set input: The input of this function is the mini-batch data that has been loaded with the enumerate() function. From these data, the tensors corresponding to positive images, negative images and the heat maps are stored on the GPU.
2. Clean the gradients: The gradients of the Adam optimizer are initialized to 0 to clear the previous gradients. This is done with PyTorch using the zero_grad() method.
3. Forward propagation step: Then there is the forward pass, where a mini-batch is passed through the network to compute the loss value.

   As has been explained, during the forward propagation step, some intermediate data can be cached to be used in the backward propagation step. This is done with the torch.set_grad_enabled() function. In order to compute the loss, the positive input samples are passed through the network until obtain the corresponding predicted heat maps ($\hat{y}$). The same is done with the negative input samples.

   Once the predicted heat maps have been computed, the total loss value is computed using the MSE criterion for both positive and negative samples, and summing both results. Finally, the results are stored for visualization.

   Furthermore, from the predicted heat maps and a threshold value, the accuracy value can be computed as explained in section 6.1.3. In this case, the TP, TN, FP and FN values for the batch are computed and, from them, the accuracy is calculated and stored for display in TensorBoard together with the loss function.
4. Backward propagation step: Finally, there is the backward pass where the new gradients are computed. This operation is performed with the backward() function.
5. Parameters update: To perform a single Adam optimization step, the step() function is called. With this function, the learnable parameters are updated based on the gradients computed, using the update rule explained in section 4.1.4.

Display visualizer train:

Then, if it is time to display the results on the TensorBoard (30s), the images are plotted with their heat maps and the scalar values of the positive loss, the negative loss, and the learning rate are plotted as explained in section 6.1.5.

Display visualizer validation:

Also, if it is time to display the results, the validation of the model should be validated. To do that, the mini-batch data is loaded using the enumerate() function. In the validation step, only one mini-batch is going to be used.

1. Set input: as for training, the tensors corresponding to positive images, negative images and the heat maps are moved on the GPU.
2. Forward propagation step: in model validation there are no gradients and no backward propagation step, since the model does not learn from the data.

   Therefore, to validate the model, the total loss value is required. In this case, the total loss is computed in the same way as in training. That is, the positive and negative input images are passed through the network to obtain the predicted heat maps. Then, the loss for positive and negative samples is computed using the MSE criterion and, finally, the total loss is computed by adding both results.

   At this step, the accuracy is also computed as in the training step.

Finally, these results are displayed in TensorBoard; the images are plotted with its heat maps and the scalar values of the positive loss and the negative loss are plotted as for training.

All of these steps are repeated for all the mini-batches to complete an epoch.

Update learning rate:

After each epoch, it is verified if the learning rate should be updated as explained above. For example, dividing it by 0,5 every 100 epochs.

Finally, in the last epoch, the matrix confusion of the model can be obtained.

This process is repeated in the different training runs. In this project, different models have been trained with different parameter values, and by visualizing the results, the model that works better has been chosen. The results obtained are shown in the next section.

6.1.5. TensorBoard and metric results

TensorBoard is the tool used to visualize the results of the training runs in order to understand and improve the model, as well as to compare different training runs with different hyper-parameters.

As previously stated, in this project, it has been decided to take advantage of two basic functionalities that TensorBoard provides: plotting scalar variables and plotting images.

Scalars values:

- Loss and accuracy:

These two terms are quite related, as both are measuring the error between the model prediction and the target. In deep learning it is important to understand these metrics in order to know how they should change during the training to know if the model is learning and, therefore, if this error is being reduced.

Consequently, it is important to plot the loss and the accuracy to see how they change during the training and validation of the model. In this case, every 30 seconds, the loss and accuracy in the current batch (for both a batch of the training set and a batch of the validation set) are visualized on the TensorBoard.

The objective is to minimize the cost function and, what is the same, maximize the accuracy. This will ensure that the model is making fewer errors. All deep learning algorithms repeat the computation many times until these metrics reach a flatter line. So if the loss decreases (and accuracy increases) over time until the curve flattens, it means that the model is learning. And the closer these lines are to 0 (for loss) and 100% (for accuracy), the better the model will work.

In addition to examining the performance of the model, these plots can provide more information, for example, if we are overfitting the model. In this case, the loss function will decrease until the curve flattens but then, at some point, the line will go up again.

Other information that can be extracted from these plots is to know if the training process is unnecessarily too long. This happens when the loss decreases rapidly and then remains stable for a long time (the flattened curve is too long).

Furthermore, TensorBoard is a great tool that allows you to visualize such metrics while the model is running. Neural networks can take hours or weeks to find a solution. But since TensorBoard updates these metrics very frequently, using it, it is not necessary to wait until the end of the training to see if the model is being trained correctly.

Finally, plotting scalars will also help to compare different training runs. In case there are different runs with different combinations of hyper-parameters, it will be useful to plot the loss and the accuracy to evaluate which combination is better. The one that approaches the

null error will be the combination that best solves the problem. This helps to improve the model, for example, by adjusting hyper-parameters.

- Learning rate:

The learning rate is also plotted to see if the learning rate decay is being done correctly.

Images:

One can think of many reasons and steps in which the image visualization can be helpful.

For example, in machine learning it is important to know what the data entering and leaving the network is like. For this reason, in this project, image plotting has been used to visualize the images that enter the neural network after undergoing transformations as well as to visualize the heat maps (predictions) that leave the neural network.

First, the input images have been plotted. This is useful for viewing the data the same way as the model does to adjust some parameters of the model. For example, plotting these images has been helpful in setting the “margin” for random cropping when preprocessing the images as explained in the previous section.

On the other hand, the other images that have been plotted are the network output activation maps superimposed on the input images. In a deep learning project for an object detection task, it is important to see if the prediction data is close the labeled data. With TensorBoard it is possible to see how this prediction changes and check if it improves over time.

Due to the current complicated situation, the results of this section could not be fully recovered. Therefore, the images that were obtained using TensorBoard could not be presented in this document.

The model has been trained with 8 different combinations of parameters (Table 6-1). In this section, using the previously defined metrics, the different training runs have been compared and the model that best fits this object detection task has been chosen.

To see what effect the parameters have on the model, one is changed on each run.

	Network image size	Cropping margin	Batch size	lr	epochs	Threshold value	Dataset loader modifications
M1	128x128	20	64	0,0002	461	-	-
M2	128x128	20	64	0,002	461	-	-
M3	224x224	20	32	0,002	261	-	-
M4	224x224	40	32	0,002	261	-	Shuffle negative samples

M5	224x224	40	32	0,002	261	0,8	Shuffle negative samples Random positive samples
M6	224x224	40	32	0,002	261	0,8	No shuffle negative samples No random positive samples
M7	224x224	40	32	0,002	261	0,8	Shuffle negative samples No random positive samples
M8	224x224	40	32	0,002	261	0,7	Shuffle negative samples No random positive samples

Table 6-1: 8 different parameters settings

First, the U-Net small has been trained using a small image size: 128x128. Between the first two training runs (M1 and M2), the difference is in the learning rate value. The first model uses a smaller learning rate. From these two training runs, the loss functions have been obtained.

As can be seen in Figure 6-28, the model is learning since the loss function decreases rapidly until the curve flattens. The loss function starts with a large number and, as the network learns, this function approaches the value 0. In this figure, in the middle of the first model training, the loss function takes a very high value. This is due to the fact that when the training process is stopped in the middle, the learning rate takes again the initial value. Consequently, the loss function has a peak, but it drops again quickly. This has been seen to always happen in these situations; by stopping the training process and reloading the data from the last epoch.

When a model is trained without interruptions, the learning rate function is seen in TensorBoard as a discrete function, being reduced to a half every 100 epochs.

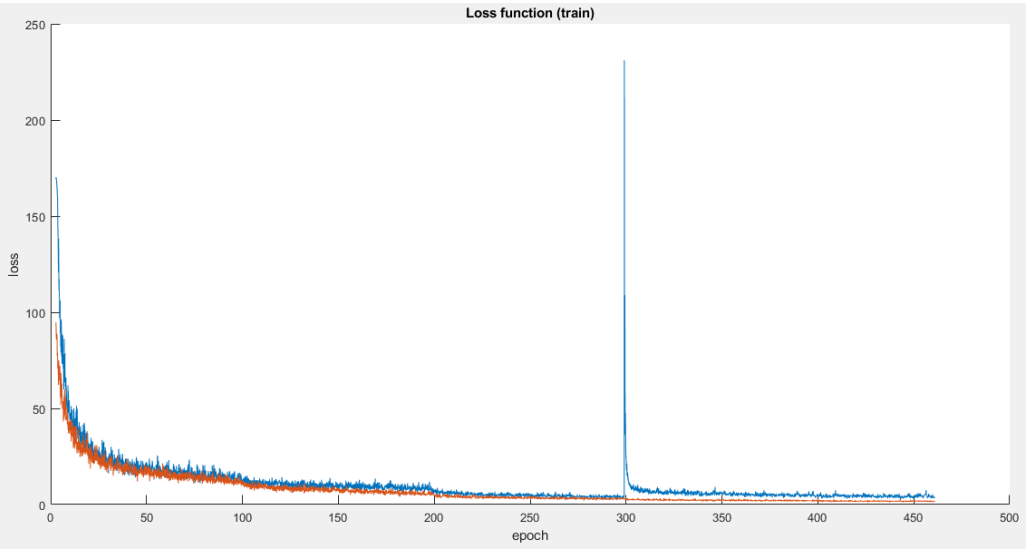


Figure 6-28:Training loss function (M1 is the function in blue and M2 is the function in red).

In order to better see the difference between these two models, the first part of the previous function has been cut out.

As previously stated, the learning rate is halved every 100 epochs. One hundred epochs have been chosen because it is where the loss function stabilizes, that is, where the curve flattens. Is at this point, where the learning rate must be reduced in order to ensure convergence. In Figure 6-29, the two loss functions decrease slightly after every 100 epochs.

As can be seen in both functions, the loss function seems to have noise. This is because it is computed on each batch rather than on each epoch. Therefore, the loss function does not decrease after each batch, but it decreases after each epoch. For this reason, the shape of the function ends up being a curve that decreases until it flattens.

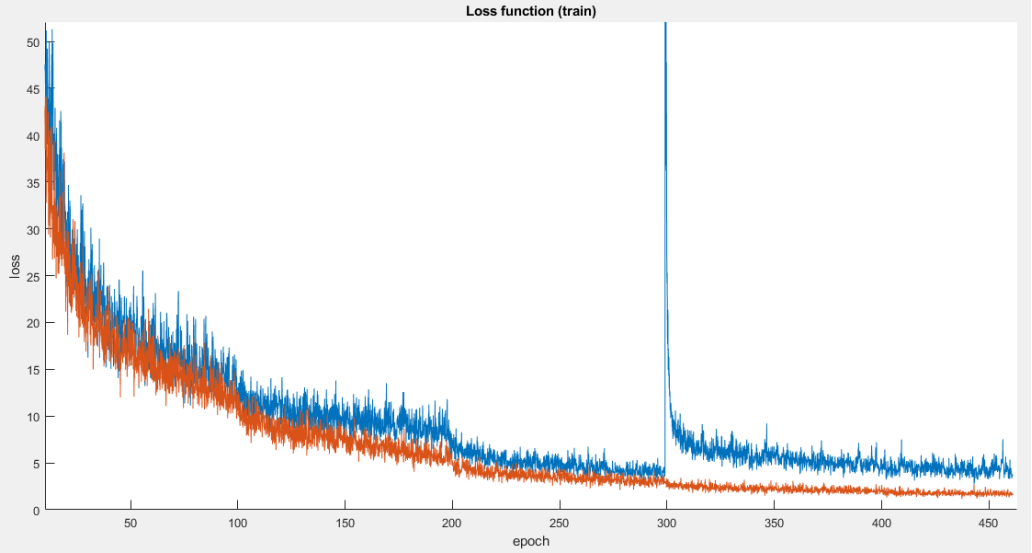


Figure 6-29: Training loss function (M1 is the function in blue and M2 is the function in red). In this case, the first part is cropped in order to zoom the part in the middle.

This figure shows a slight difference between the models. Considering that the blue line corresponds to M1 loss function and the red line corresponds to M2 loss function, the second one is better since is closer to the value 0. However, the difference between the validation loss functions is more important. These functions are plotted in Figure 6-30.

In the figure, it can be seen that both functions are similar. As in the training process, the validation loss function also decreases until it flattens. In terms of validation loss, both models perform equal. It does not matter which model, or what is the same, which learning rate value is chosen, since in this case there is no difference.

Whereas in the training process, the loss function ends near a loss value equal to 2, in the validation process, the function ends above a loss of 20. As stated in the previous sections, this is acceptable because the model learns from the training data and is validated using another data. In addition, we are not overfitting the model because the loss function decreases

and does not rise again. There is no overfitting because many regulation measures are introduced into the model.

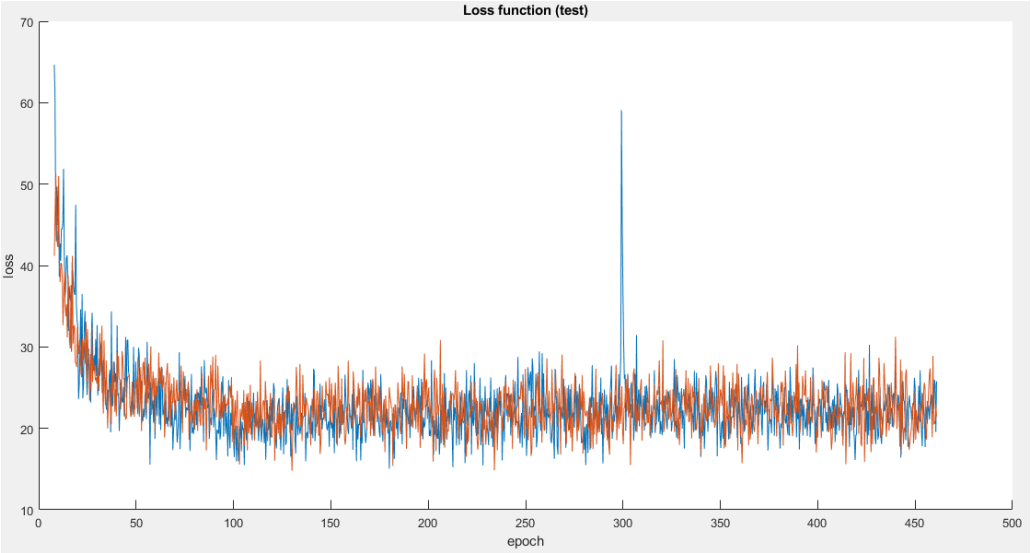


Figure 6-30: Training loss function (P1 is the function in blue and P2 is the function in red).

According to the loss graph, you can stop training the model before 461 epochs, since after the epoch 100 it does not improve much.

Then, the learning rate value of the second model (0,002) is taken to use it in the third model. This value has been chosen because, although the two models end up working similarly with the validation set, the second model works slightly better with the training set.

As mentioned above, the model can be trained for less than 461 epochs. In this case, 261 epochs have been chosen. In addition, a 128x128 size image is a too small image. Many researchers use higher image sizes, for example the 224x224 size is typical. If possible, is better to train the network with larger images so that later it can also work well in real cases where the images will usually be larger.

In this case, as the images are larger, a mini-batch contains a lot of data. For this reason, so that the network can be trained without saturating the GPU, it is important to decrease the batch size (from 64 to 32).

Figure 6-31 shows the two loss functions of the two models M2 and M3. The blue function corresponds to the M3 loss function. This function also has a peak due to stopping and loading the training process. The red function corresponds to the M2 loss function.

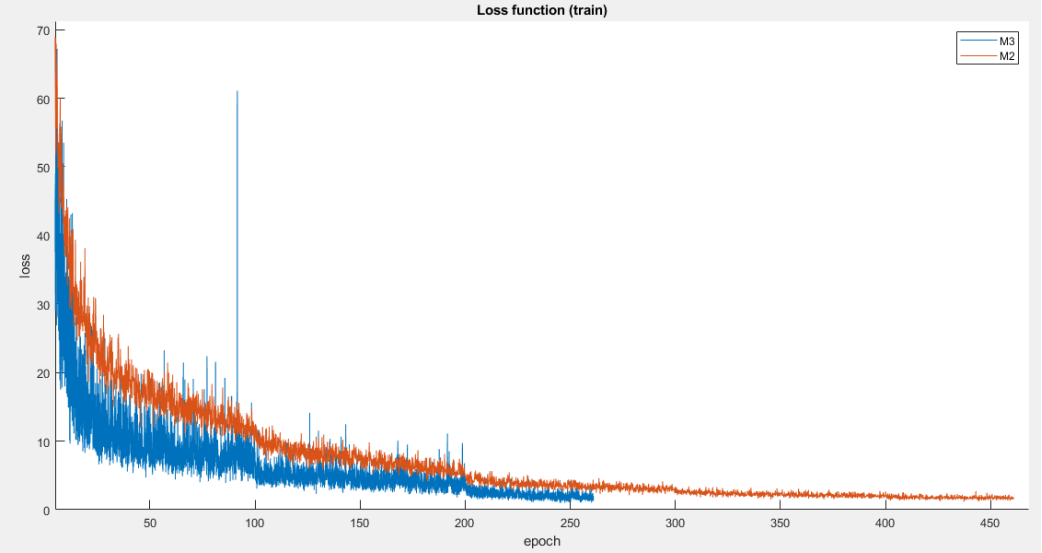


Figure 6-31: Training loss function (M3 is the function in blue and M2 is the function in red).

As can be seen in this figure, the blue loss function goes below the red function. The reason for this is due to the batch size. The new model (M3), is trained with small batches. Since the loss value represent the batch loss, is accepted the loss function of this new model (M3) to be below the other one (M2). It does not mean that the model works better.

The same is seen in Figure 6-32.

However, it is possible to see that the loss function decreases and stabilizes after 100 epochs. Therefore, the number of epochs is sufficient for the model to learn the parameters.

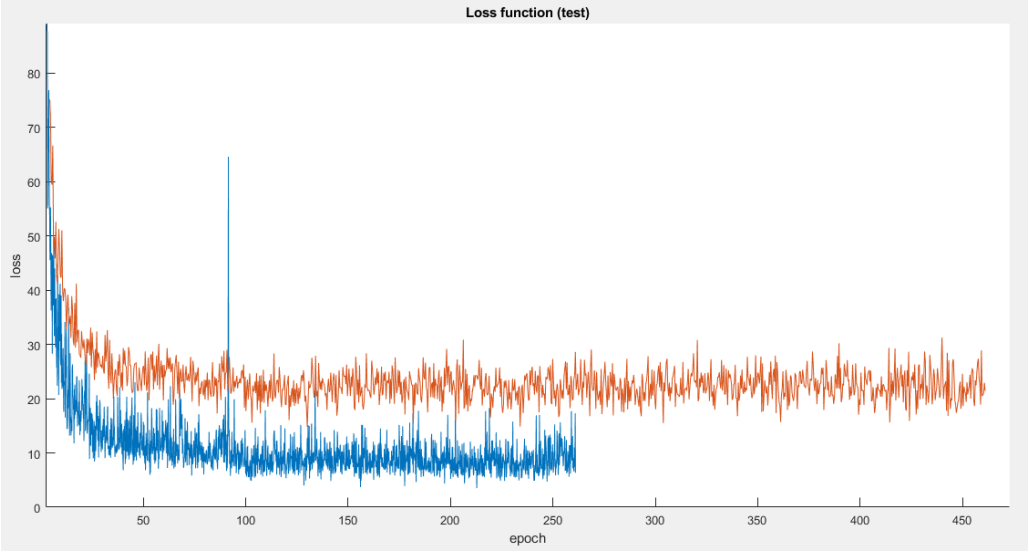


Figure 6-32: Validation loss function (M3 is the function in blue and M2 is the function in red).

Since it is possible to work with larger images even in real time with the Jetson AGX Xavier, the larger network image size (224x224) is chosen.

Then, this image size is used in the fourth model. This model has the same parameters as model 3, but another "margin" value. This parameter was explained in section 6.1.4.1.1 and it is important to set it to the correct value so as not to crop the keypoint region.

Using TensorBoard, the network input images have been shown. In some cases, by randomly cropping the images, as the margin parameter is too small, part of the region around the keypoint is cropped. Therefore, in this case, these images are like negative samples but the model takes them as positive. For this reason, in model 4 (M4), this parameter has been set to 40 instead of 20 pixels. By displaying the input images again, it has been possible to see that this problem no longer appears.

Furthermore, the negative samples have been randomly shuffled before being splitted into the training and validation set. If they are not reshuffled, the images are sorted according to the order in which they were collected.

Finally, 4 more models are trained. In these models, the way to create a sample of the dataset is changed. This means changing the __getitem__() function. Three different combinations have been tested. In addition, the threshold value has been added to compute the evaluation metrics. As a remainder, it is possible to see the accuracy function in TensorBoard and the predicted heat maps to see their evolution. It has been seen that the form of the accuracy function is the expected one as well as the predictions, which are increasingly more accurate.

First, in model 5 (M5), the negative samples are randomly reshuffled and each index is chosen in a random way within a range. Here, the threshold value is equal to 0,8.

The results are shown in Table 6-2 and Table 6-3. These tables are the confusion matrix computed in the last epoch, in the validation and in the training processes respectively. Figure 6-33 and Figure 6-34 show different FN and FP errors in validation. And Figure 6-35 and Figure 6-36 show different FN and FP errors in training.

Confusion matrix (M5) Validation		**Target**			
		Positive	Negative		
Model	Positive	TP=285	FP=1	Positive Predicted Value	0,9965
	Negative	FN=3	TN=287	Negative Predicted Value	0,9897
		Sensitivity	Specificity	**Accuracy** = 0,9931	
		0,9896	0,9965		

Table 6-2: Confusion matrix and metrics (M5, validation)

Figure 6-33: False negatives in validation (M5)

Figure 6-34: False positives in validation (M5)

Confusion matrix (M5) Training		**Target**			
		Positive	Negative		
Model	Positive	TP=2711	FP=10	Positive Predicted Value	0,9963
	Negative	FN=9	TN=2710	Negative Predicted Value	0,9967
		Sensitivity	Specificity	**Accuracy** = 0,9965	
		0,9967	0,9963		

Table 6-3: Confusion matrix and metrics (M5, training)

Figure 6-35: False negatives in training (M5)

Figure 6-36: False positives in training (M5)

After training model 5, it has been seen that both samples (positive and negative) were repeated in the same epoch. This happens because in the __getitem__ function, the sample index is chosen randomly instead of having one index per sample. Therefore, in model 6, this

is changed in order to take all the different positive samples of the training set to train the network instead of repeating them.

On the other hand, the negative samples are not reshuffled to also see its effect. These results are shown in Table 6-4 and in Table 6-5. Figure 6-37 shows different FN errors in validation. And Figure 6-38 and Figure 6-39 show different FN and FP errors in training.

Confusion matrix (M6) Validation		**Target**			
		Positive	Negative		
Model	Positive	TP=278	FP=0	Positive Predicted Value	1,0000
	Negative	FN=10	TN=288	Negative Predicted Value	0,9664
		Sensitivity	Specificity	**Accuracy** = 0,9826	
		0,9653	1,0000		

Table 6-4: Confusion matrix and metrics (M6, validation)

Figure 6-37: False negatives in validation (M6)

Confusion matrix (M6) Training		**Target**			
		Positive	Negative		
Model	Positive	TP=2708	FP=3	Positive Predicted Value	0,9989
	Negative	FN=12	TN=2717	Negative Predicted Value	0,9956
		Sensitivity	Specificity	**Accuracy** = 0,9972	
		0,9956	0,9989		

Table 6-5: Confusion matrix and metrics (M6, training)

Figure 6-38: False negatives in training (M6)

Figure 6-39: False positives in training (M6)

Model 7 (M7) uses the same threshold value. In this case, the negative samples are randomly reshuffled again. These results are shown in Table 6-6 and in Table 6-7. Figure 6-40 and Figure 6-41 show different FN and FP errors in validation. And Figure 6-42 and Figure 6-43 show different FN and FP errors in training.

Confusion matrix (M7) Validation		**Target**			
		Positive	Negative		
Model	Positive	TP=286	FP=2	Positive Predicted Value	0,9931
	Negative	FN=2	TN=286	Negative Predicted Value	0,9931
		Sensitivity	Specificity	**Accuracy** = 0,9931	
		0,9931	0,9931		

Table 6-6: Confusion matrix and metrics (M7, validation)

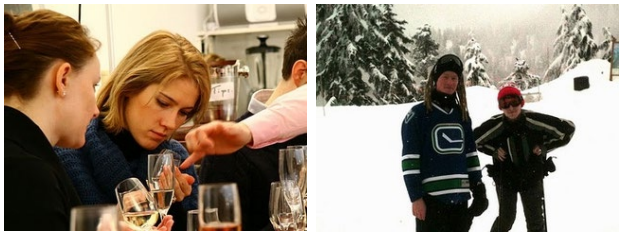


Figure 6-40: False negatives in validation (M7)

Figure 6-41: False positive in validation (M7)

Confusion matrix (M7) Training		**Target**			
		Positive	Negative		
Model	Positive	TP=2714	FP=21	Positive Predicted Value	0,9923
	Negative	FN=6	TN=2699	Negative Predicted Value	0,9978
		Sensitivity	Specificity	**Accuracy** = 0,9950	
		0,9978	0,9923		

Table 6-7: Confusion matrix and metrics (M7, training)

Figure 6-42: False negatives in training (M7)

Figure 6-43: False positives in training (M7)

	Accuracy		FP		FN	
M5	0,9931	0,9965	1	10	3	9
M6	0,9826	0,9972	0	3	10	12
M7	0,9931	0,9950	2	21	2	6

Table 6-8: Results of the models M5, M6 and M7

Table 6-8 shows the results obtained with the previous models grouped; validation results in red and training results in black. Although the validation results are used to compare the different models, the training results have also been shown to see what the network is learning.

The **M6 model** is the one with the worst validation results in terms of precision. As can be seen in Table 6-8, there are many false negatives. This means that positive samples are classified as negative. By looking at all the samples where the algorithm fails, it can be seen that there are samples that are clearly positive (for example, images of front faces). Also, it can be seen that the algorithm easily fails with images of people at a far distance.

As this model fails a lot in the face detection (there are many FN) and gives a lower precision than the other models, this model is discarded.

Model 5 (M5) is better than M6. In this model, negative images are randomly reshuffled before being splitted into the two datasets, and then selected using a random index. What happens here is that when showing the images that enter the network, there are images that are repeated in the same epoch.

In this case, there are almost the same cases of FP and FN. Although it is quite accurate, we want all the images in the dataset to be taken and not repeated. When creating the dataset, different types of samples have been taken and it has already been ensured that they are in the same proportion. If the samples are repeated this no longer makes sense.

As can be seen in Figure 6-33 and in Figure 6-35, this algorithm fails with people in profile views or when the person is at a great distance. However, in Figure 6-36, it can be also seen that the algorithm fails with people in profile. This is due to the fact that when labeling the dataset, there have been doubts with the samples of half-profile people where part of the region around the eye is also seen; it has been difficult to know if they were positive or negative samples. Therefore, seeing that there are similar images labeled as positive and negative samples, it is likely that the algorithm will fail with these samples and classify them backwards.

Model 7 (M7) is better than the previous two. In this model, negative images are randomly shuffled before being separated, and images are selected in order, not with a random index. This ensures that there are no repeated samples. Also, if you want a random order, this can be done with the "shuffle" argument of the dataloader. In this way, model 7 can learn all the diversity of images that exist in the dataset, both positive and negative images (people from near, far, in profile, from behind, etc.).

This model achieves an accuracy just as good as in the previous case and at the same time it uses the entire dataset. This is true for validation, since accuracy in training decreases as false positives increase. As can be seen in Figure 6-41 and Figure 6-43, these false positives correspond to images of people in profile. However, in Figure 6-40 and in Figure 6-42, the algorithm also fails with samples of people in profile. These samples correspond to the FN.

In this case, there is a slightly difference between these two errors (FP and FN). In the case of the false negatives, the samples are images of people in profile but at a great distance. On the other hand, in the case of the false positives, the samples are images of people in profile but the keypoint region can be seen more easily than in the previous cases. It looks like the FPs could be TP. This shows that the labeling of the profiles has not been precise enough. However, in this case, it seems that the algorithm learns better, since it labels the clear profiles as positive, and when the person is farther away, it leaves them as negative samples.

In this case, the FN decrease with respect to the previous models. This means that the model's sensitivity increases. Therefore, the algorithm tends to identify positive samples well as positive. Even so, it is only seen that the algorithm has found a probability greater than the threshold value; it is not seen if the location of the keypoint is correct.

Finally, the same parameters combination is used in **model 8** (M8). In this case, the threshold value is decreased to 0,7 in order to make it easier to detect people who are at a greater distance.

The results are shown in Table 6-9 and in Table 6-10. Figure 6-44 and Figure 6-45 show different FN and FP errors in validation. And Figure 6-46 and Figure 6-47 show different FN and FP errors in training.

Confusion matrix (M8) Validation		Target			
		Positive	Negative		
Model	Positive	TP=286	FP=2	Positive Predicted Value	0,9931
	Negative	FN=2	TN=286	Negative Predicted Value	0,9931
		Sensitivity	Specificity	**Accuracy** = 0,9931	
		0,9931	0,9931		

Table 6-9: Confusion matrix and metrics (M8, validation)

Figure 6-44: False negatives in validation (M8)

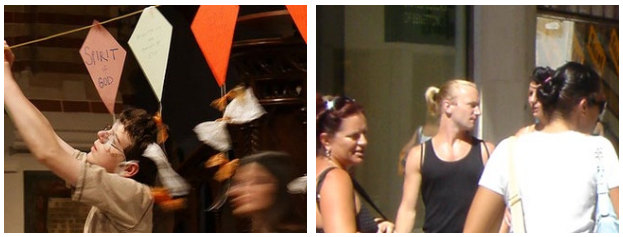


Figure 6-45: False positives in validation (M8)

Confusion matrix (P8) Training		Target			
		Positive	Negative		
Model	Positive	TP=2719	FP=14	Positive Predicted Value	0,9949
	Negative	FN=1	TN=2706	Negative Predicted Value	0,9996
		Sensitivity	Specificity	**Accuracy** = 0,9972	
		0,9996	0,9949		

Table 6-10: Confusion matrix and metrics (M8, training)

Figure 6-46: False negatives in training (M8)

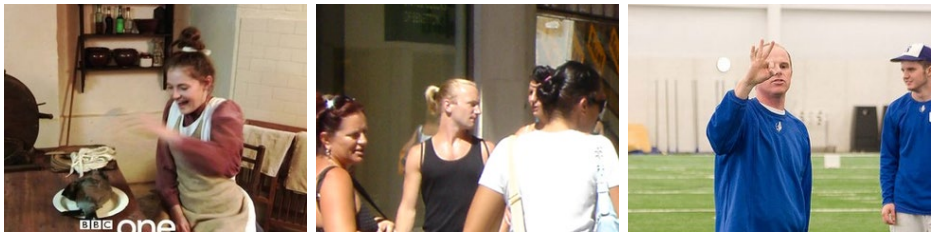


Figure 6-47: False positives in training (M8)

With this lower threshold value, equal precision is achieved for model validation but better in training. This happens because FN and FP decrease. That is, there are fewer errors.

In this case there are also more FP than FN. However, as can be seen in the figures above, the algorithm fails less with images of people at a greater distance. For example, as shown in Figure 6-44, the only images in which it fails to detect positives (FN), are two images in which it has been seen that the algorithm always fails. On the other hand, the FP are images that contain people in profile where part of the keypoint region can be seen.

It has been decided to choose this model for the rest of the project, since it gives a good accuracy value and it can detect people that is far away. However, it will tend to consider images of people in profile as positive samples.

More parameters and hyper-parameters can be changed and tested. However, in this project it is assumed that this accuracy values are good enough to implement the algorithm in real time together with the depth extraction.

6.1.6. Test the model

Once the best model is chosen (Annex 12.4), the learned model and its parameters are used to verify its performance with different images.

To test the model, some videos have been recorded with the ZED camera. In this case, this camera is used like a regular camera and the stereo mode is not used. This camera has a two types of resolution available (Table 6-11).

Video mode	Frames per second	Output resolution (side by side)
1080p	30	3840x1080
720p	60	2560x720

Table 6-11: ZED camera video features

For example, when using the 1080p, the output video resolution is 3840x1080. This is because each frame contains two images attached, the left RGB image and the right RGB image (Figure 6-48). The image on the left is the one that passes through the network.

Figure 6-48: ZED video frame

It is important to test the model with images taken with the camera that is going to be used in a real case. There are different types of cameras used in robots, and the ZED camera is one of them. The images taken with this camera will have another characteristics as the images in the dataset (different image quality, resolution, etc.). Here is where it will be seen if the deep learning model works.

Then, some scripts (Annex 12.5) are used to read the video, process its frames and create a video with the processed frames.

First, the video is read and every 3 frames, one is processed. This frame is preprocessed and passed through the network. Then, from the predicted output (heat map), the coordinates of the point with the maximum probability (keypoint) and its probability value are taken. This probability value is compared with the threshold value. The threshold value used is 0,7. If the probability is greater than the threshold, a circle is drawn at the keypoint coordinates in the processed frame. Finally, this frame will be inserted in the output video.

After testing the model using different videos, the results and conclusions are the following:

- **True negative samples:**

In the case of negative samples, the algorithm gives very good results. The probability in these frames is less than 0,10. Among these images are:

- Images without people (Figure 6-49):

Figure 6-49: Video frame. Resolution 1080p. Probability=0,02. Detection time=24ms

- Images with people but its face outside the frame (Figure 6-50):

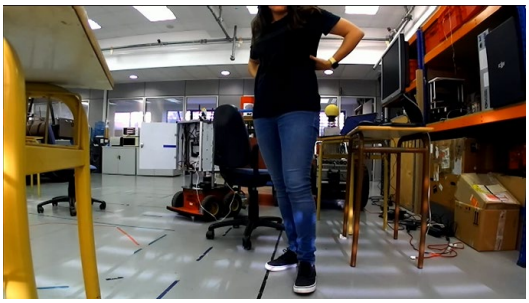


Figure 6-50: Video frame. Resolution=720p. Probability=0,02. Detection time=24ms

- Images with people but its face outside the 224x224 region (Figure 6-51):

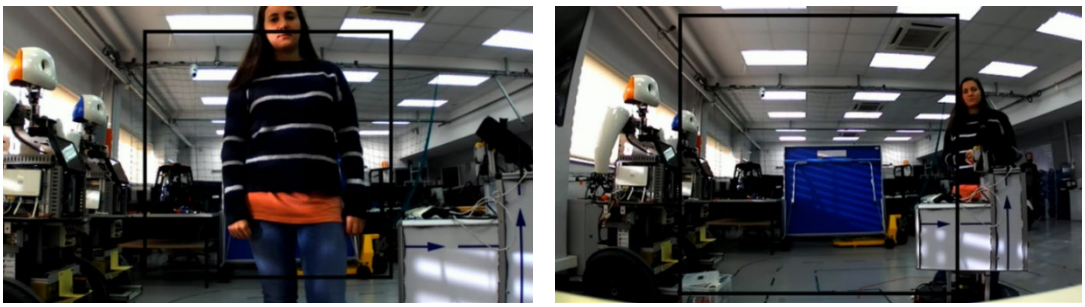


Figure 6-51: Video frame. On the left, resolution=1080p and probability=0,05 and on the right, resolution=720p and probability=0,02. Detection time=24ms.

- Images with people from behind (Figure 6-52):

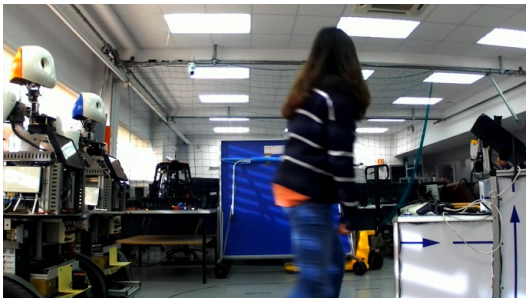


Figure 6-52: Video frame. Resolution=1080p. Probability=0,04. Detection time=24ms.

- Images of people completely in profile (Figure 6-53):

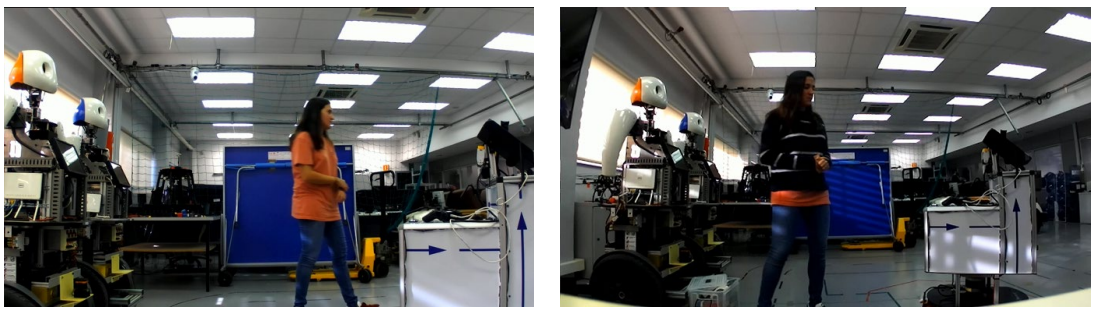


Figure 6-53: Video frame. On the left, resolution=1080p and probability=0,02 and on the right, resolution=720p and probability=0,03. Detection time=24ms

- Images where the eyes can't be seen due to an external element (illumination conditions):

  For example, in the image in Figure 6-54, even for a human, it will be almost impossible to detect the keypoint in the correct place.

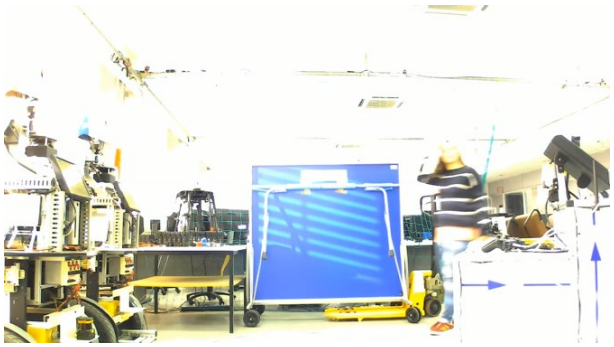


Figure 6-54: Video frame. Resolution=1080p. Probability=0,02. Detection time=24ms

- **True positive samples:**

In the case of positive samples, the algorithm gives also good results. When the person is at a sufficient distance to be detected, the algorithm detects it with precision. At short distances, the algorithm almost always gives probabilities greater than 0,9. As the person moves away, the probability decreases until it falls below the threshold value.

All this is true for the highest resolution (1080p). When using the resolution 720p, the algorithm fails more often.

Below, it is verified that what was searched for when the dataset was created is met.

- Different illumination conditions:

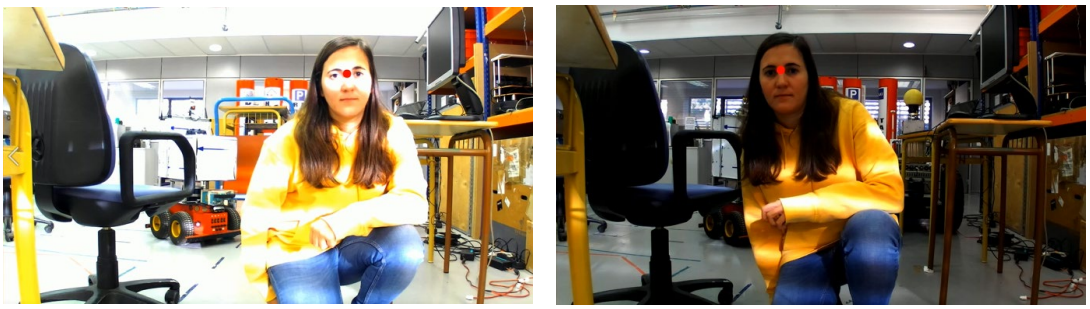


Figure 6-55: Video frames. Resolution=1080p. On the left, probability=0,99 and on the right, probability=0,96.

- Blurred images:

Figure 6-56: Video frame. On the left, resolution=1080p and probability=0,98 and, on the right, resolution=720p and probability=0,98. Detection time=24ms

- Different perspective:

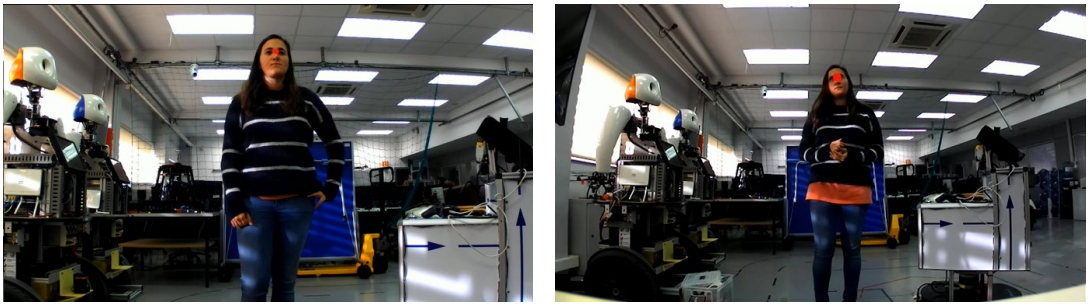


Figure 6-57: Video frame. On the left, resolution=1080p and probability=1,00 and, on the right, resolution=720p and probability=0,72. Detection time=24ms.

- Different head orientation:

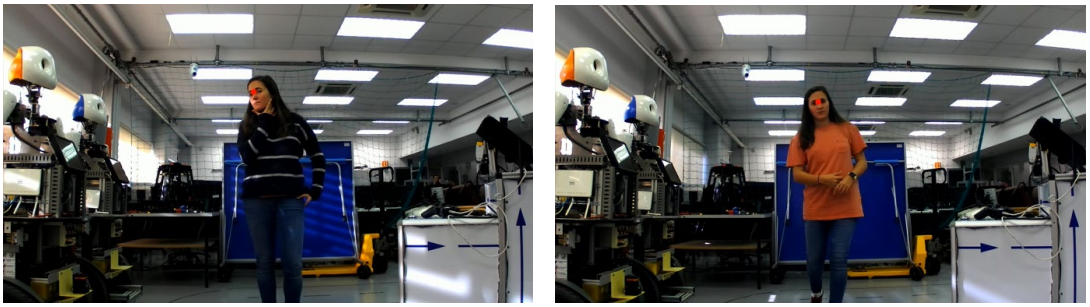


Figure 6-58: Video frame. Resolution=1080p. On the left, probability=0,94 and on the right, probability=1,00.

- Closed eyes:

Figure 6-59: Video frame. Resolution=1080p. Probability=0,92. Detection time=24ms

Figure 6-60: Video frame. Resolution=1080p. Probability=0,99. Detection time=24ms

- Different distances:

Below, two image sequences (one for 1080p resolution and one for the 720p resolution) where the person is moving away.

1080p:

Figure 6-61: Video frame. Resolution=1080p. Probability=0,97. Detection time=24ms

Figure 6-62: Video frame. Resolution=1080p. Probability=1,00. Detection time=24ms.

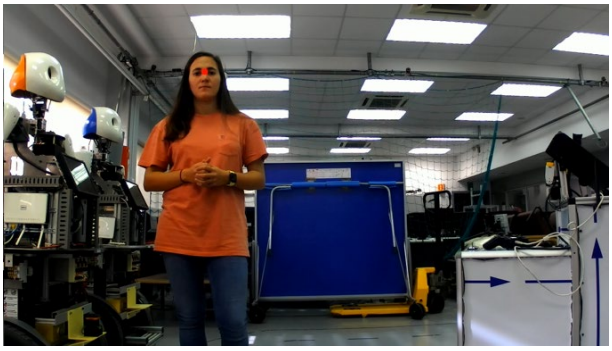


Figure 6-63: Video frame. Resolution=1080p. Probability=1,00. Detection time=24ms

Figure 6-64: Video frame. Resolution=1080p. Probability=1,00. Detection time=24ms

Figure 6-65: Video frame. Resolution=1080p. Probability=1,00. Detection time=24ms

Figure 6-66: Video frame. Resolution=1080p. Probability=0,92. Detection time=24ms

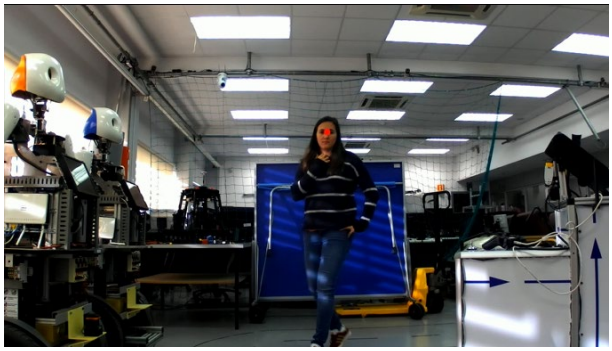


Figure 6-67: Video frame. Resolution=1080p. Probability=0,87. Detection time=24ms.

720p:

Figure 6-68: Video frame. Resolution=720p. Probability=0,98. Detection time=24ms

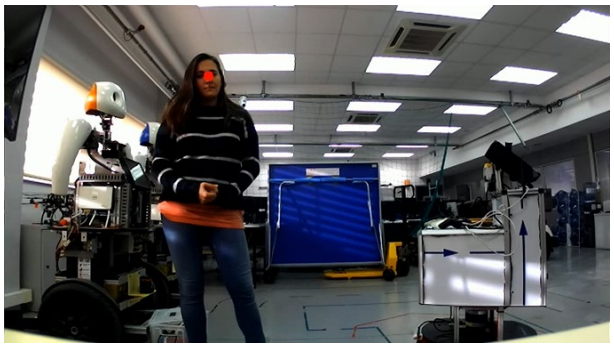


Figure 6-69: Video frame. Resolution=720p. Probability=0,94. Detection time=24ms

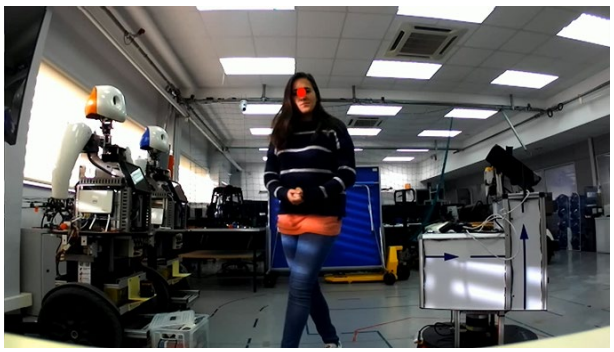


Figure 6-70: Video frame. Resolution=720p. Probability=0,91. Detection time=24ms

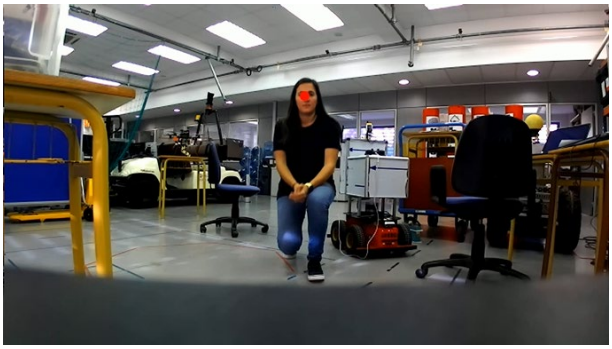


Figure 6-71: Video frame. Resolution=720p. Probability=0,94. Detection time=24ms

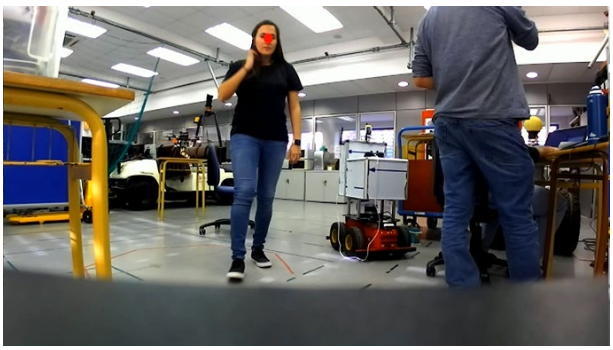


Figure 6-72: Video frame. Resolution=720p. Probability=1,00. Detection time=24ms

Finally, it is concluded that thanks to the careful preparation of the dataset, the algorithm is capable of detecting the person under all these conditions, even when they are unfavourable.

- **False negative samples:**

In this case, a difference between the resolution of 720p and 1080p must be done.

As previously said, when using the 1080p resolution, the algorithm is more precise. When the person is approximately close than 4m, it can detect the person most of the time. Therefore, if the lighting conditions, the person's movement, its posture, etc. allow the keypoint region to be seen (as in the previous cases), the algorithm will detect the person's face.

However, in one situation the algorithm sometimes fails. This is the case of images where the person's face is within the region but on the edge (Figure 6-73). Although the probability (0,41) is higher than in the negative samples, it is not sufficient to identify it as a positive sample.

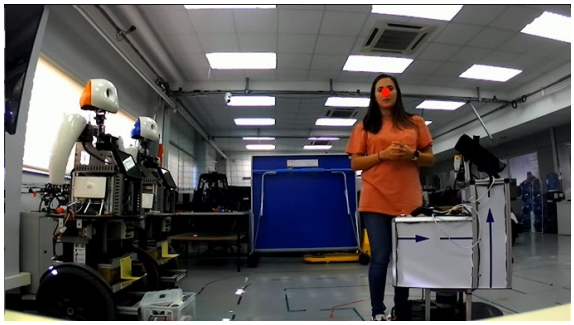
Figure 6-73: Video frame. Resolution=1080p. Probability=0,41. Detection time=24ms

In the next figure (Figure 6-74), it is even more complicated for the algorithm to detect the person.

Figure 6-74: Video frame. Resolution=1080p. Probability=0,17. Detection time=25ms

Even so, other times the algorithm succeed.

On the other hand, when using 720p resolution, the algorithm is less accurate and it is possible that in one frame it will detect the person, but in the next frame, even if the person is in the same place, it will not detect it (Figure 6-75, Figure 6-76, Figure 6-77, Figure 6-78).



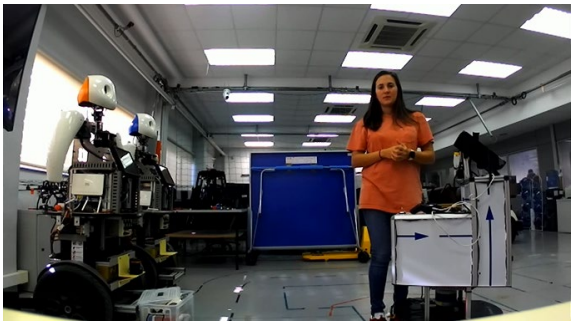
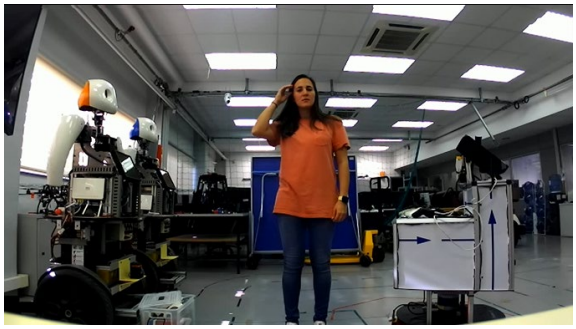


Figure 6-75: Video frames. Resolution=720p. On the left, probability=0,71 and on the right (next frame), probability=0,39.



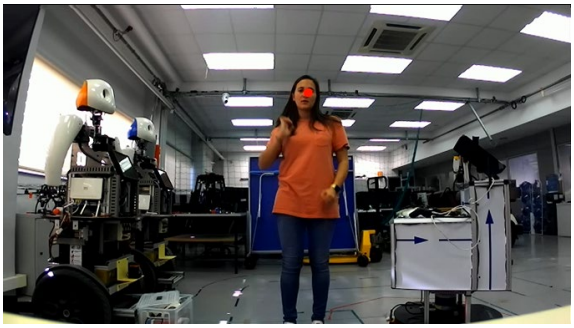


Figure 6-76: Video frames. Resolution 720p. On the left, probability=0,48 and on the right (next frame), probability=0,95.

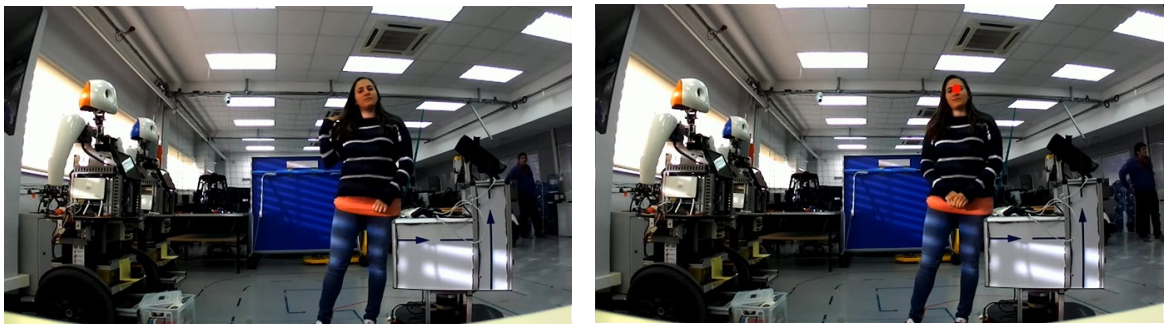


Figure 6-77: Video frames. Resolution 720p. On the left, probability=0,50 and on the right (next frames), probability=0,86.

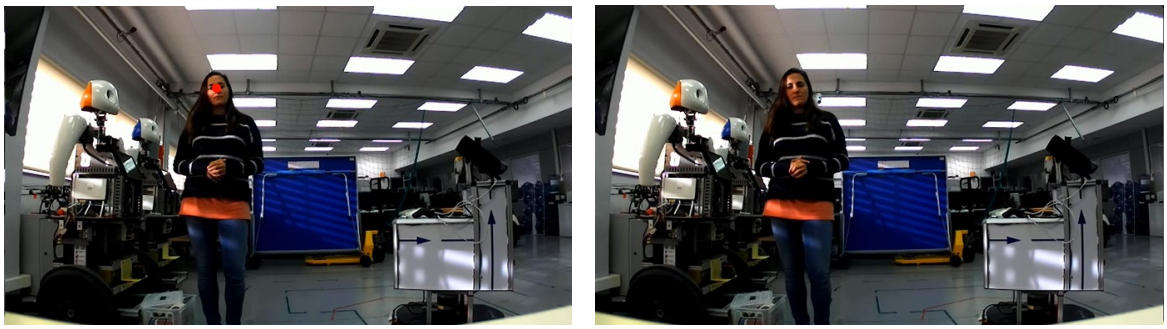


Figure 6-78: Video frames. Resolution=720p. On the left, probability=0,97 and on the right (next frames), probability=0,27.

Considering that a person is within the area where the algorithm is able to detect, as can be seen, when using the 720p resolution, intermediate probability values are obtained. Conversely, when using the 1080p resolution, the probabilities are usually between 0 and 0,1 for the negative samples and between 0,7 and 1 for the positive samples.

In specific cases such as in the previous figures or when people are half-profile, intermediate values may appear. But this is not common.

Finally, for both resolutions, when the person is far, the algorithm cannot detect the person (Figure 6-79 and Figure 6-80).

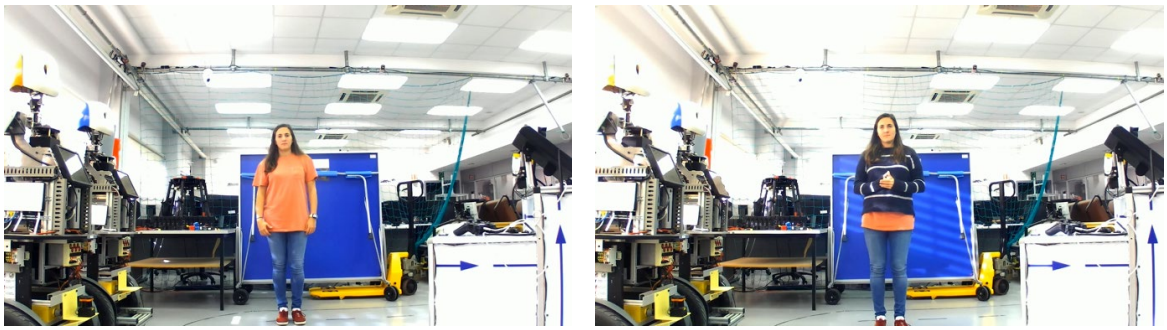


Figure 6-79: Video frames. Resolution=1080p. On the left, probability=0,13 and on the right, probability=0,34.

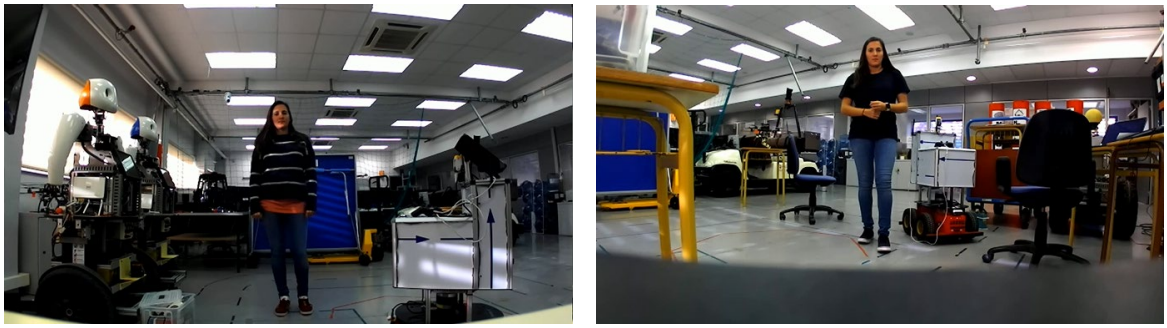


Figure 6-80: Video frames. Resolution=720p. On the left, probability=0,02 and on the right, probability=0,26.

- **False positive samples:**

As seen in the previous section, most of the false positive samples were images of half-profile people. During the dataset labeling, there were different samples that were difficult to decide if they were positive or negative samples. This would depend on the final application and how the robot behaviour is programmed.

Figure 6-81, shows an image that contains a people's face, where one eye and part of the other can be seen. In this case, the algorithm identifies it as positive. It has been seen that, as it learns the region between the eyes and these types of images contains part of the region, usually the algorithm identifies them as positive. Also, the algorithm locates the keypoint well.

Figure 6-81: Video frame. Resolution=1080p. Probability=0,98. Detection time=24ms

On the other hand, Figure 6-82 shows an image of a person looking at the ground. In this case, the decision is also difficult. In this case, it seems logical because the dataset contains images with people with closed eyes. So the algorithm also learns this and, for this reason, it detects the people's face. In addition, it locates the keypoint quite well.

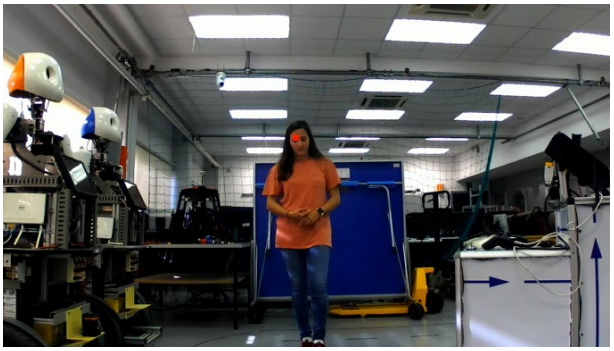


Figure 6-82: Video frame. Resolution=1080p. Probability=0,71. Detection time=24ms

At the end, this should be evaluated and considered if it would or not be a problem. If not, all remains equal, but otherwise, some changed should be done, for example, in the dataset by including these types of images as negative samples.

Consequently, it is difficult to say if they are true positive samples or false positive samples.

Finally, in addition to all of this, the three important differences between the 1080p resolution and the 720p resolution are:

- Wasted information:

On one hand, the 1280x720 images are downscaled to a 1/3 (460x240). In this case, when cropping the image to fit the network size, some information is lost: the information of 8 pixels above and below, and the information of 101 pixels on the right and on the left (Figure 6-83).

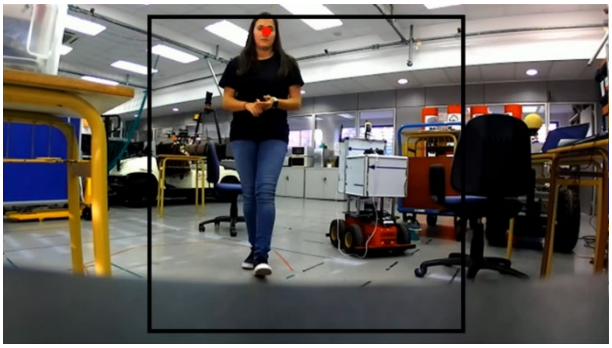


Figure 6-83: Video frame. Resolution=720p. Probability=0,88. Detection time=24ms

On the other hand, 1920x1080 images are downscaled to a 1/4 (480x270). In this case, when cropping the image to fit the network size, some information is lost: the information of 23 pixels above and below, and the information of 128 pixels on the right and on the left (Figure 6-84).

Figure 6-84: Video frame. Resolution=1080p. Probability=0,03. Detection time=24ms

Therefore, using the 720p resolution, less information is wasted because the part of the image that is passed to the network is proportionally bigger than when using the 1080 resolution. For this reason, when using the 720p resolution, is easier to detect the person if it is close to the camera and above it.

- Distance:

After recording and reviewing the videos, it seems that the algorithm with 1080p resolution detects a person up to 4m, while with 720p resolution it detects a person up to approximately 2,5m. This will be verified when stereo vision is used to extract the distance value.

Below, Figure 6-85 shows the maximum distance detected using 1080p resolution and Figure 6-86 shows the maximum distance detected using 720p resolution.

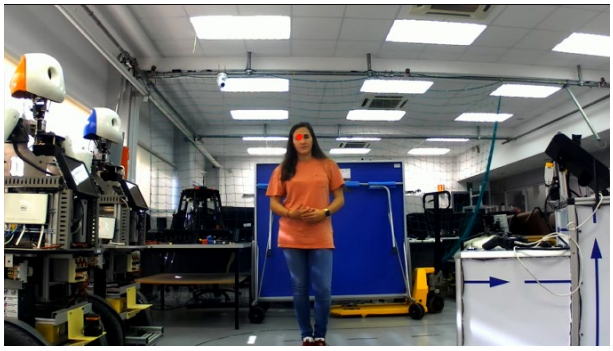


Figure 6-85: Video frame. Resolution 1080p. Probability=0,83. Detection time=24ms

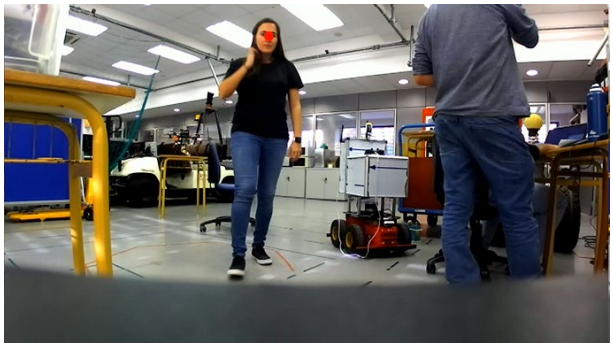


Figure 6-86: Video frame. Resolution=720p. Probability=1,00. Detection time=24ms

- Precision:

As can be seen in figures from Figure 6-75 to Figure 6-78, when using the 720p resolution, the algorithm fails more often, even if the person is close to the camera.

In Annex 12.6, more images can be found.

Finally, as can be seen, the network takes 24ms to process a frame. This means that in one second, 40 frames could pass through the network. To this we will have to add the preprocessing time of the image but, it can be said that the algorithm can work in real time.

6.2. Stereo Vision

Once the deep learning process has been completed and it has been proven that it detects people well, the distance is extracted using stereo vision. In this way, it will be possible to locate a person in the 3D world.

To do it, the trained parameters and the NN model are stored in the Jetson AGX Xavier. Then, the ZED camera is connected to it and initialized using a ROS node. This ROS node provides access to the left and right rectified images, the depth map, the 3D point cloud, etc. Also, in this ROS node there is a file with all the parameters of the camera such as the resolution, the frame rate, the quality, the minimum and maximum depth, etc. These parameters can be configured by modifying this file.

Therefore, the ZED is available in ROS as a node that published its data to topics [46]. The topics that have been used in this project are:

- rgb/image_rect_color: color rectified image (left RGB image by default)
- depth/depth_registered: depth map image registered on left image.

The first topic corresponds to the left RGB rectified image.

Stereo image rectification projects images onto a common image plane in such a way that the corresponding points have the same row coordinates [47]. The advantage of this is that computing stereo correspondences is simplified to a 1-D search problem along the rows of the rectified images (Figure 6-87). Therefore, this image projection makes the images appear to be taken with parallel cameras.

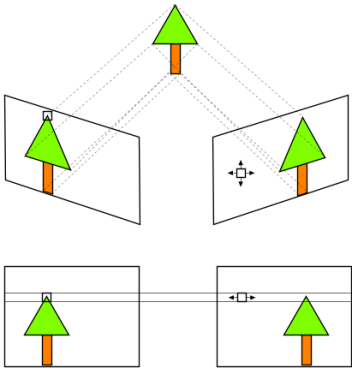


Figure 6-87: Image rectification [48]

The deep learning algorithm uses this image to detect the keypoint between the person's eyes. The left image is used since it holds the reference system.

The second topic is the depth map. This map stores a distance value (z) for each pixel (x,y) in the image. The depth is registered on left image, that is, it has been computed with respect to the left image. Therefore, the (x,y) pixel in depth map corresponds the (x,y) pixel in the left rectified image. This makes it easier to find the depth coordinate of the detected keypoint. The distance in the depth map is expressed in metric units (meters, in this case).

The depth map will be displayed as a grayscale image, where each tone of gray represent a different distance value. The brightest pixels represent the most distant possible depth value and the darkest pixels represent the closest possible depth value.

Although the minimum and maximum depth parameters of the camera may be 0,3m and 20m, these parameters are set to 0,7m and 10m, respectively. These values have been chosen because if the algorithm detects people at 4m maximum, there is no point in having a more complex depth map with points up to 20 m. In this case, values that are greater than 10m will be set to infinity. The same is done with the minimum depth. The value of 0,7m has been considered as a reasonable distance value, since the robot does not need to get as close to the person, especially knowing that part of the robot will be ahead of the camera position. In this case, values that are smaller than 0,7 m will be set to minus infinity. If the range is wider (0,3 and 20m), it requires more memory.

As previously said, a ROS node is required to communicate the camera with the deep learning algorithm in the Jetson AGX Xavier.

6.2.1. ROS node

The ROS node is programmed using Python (Annex 12.7.1).

First, it is required to determine the types of messages that the topics will contain.

- Image message: this message contains an uncompressed image. The (0,0) is at top-left corner of image.
- Float 64 message: this message contains float data.

The Image message is used for topics that contains images such as the ZED topics of the left RGB rectified image and the depth map, as well as the topic used to publish the neural network resulting image. To work with images, the OpenCV library is used. The OpenCV (Open Computer Vision) is a library mainly aimed at real-time computer vision.

On the other hand, the Float 64 message is used to publish the depth value and the times of the different processes.

Node initialization:

First, the object detector is defined. This part of the code contains the deep learning model initialization.

- Probability threshold: the threshold value is set here.
- Test options: all the test options (hyper-parameters and other options necessary to use the model trained) are loaded.
- Image transformations: the list of transformations to apply at the images are defined here. This function will transform the OpenCV images to PyTorch tensors and will also normalize them.
- Model: the trained model is loaded.

Then, a bridge is created between ROS images and OpenCV images. This is required because the images are received as ROS image messages, but must be manipulated using OpenCV.

Next, the ROS subscribers and publishers are defined.

- Depth map: this node is subscribed to the ZED camera depth map topic (depth/depth_registered).
- RGB image: this node is subscribed to the ZED camera left RGB rectified image topic (rgb/image_rect_color).
- Resulting image: this node publishes the resulting RGB image with the detected keypoint, the probability, the process time and the distance value.
- Depth value: this node publishes the resulting depth value. That is, the distance in meters of the detected keypoint.

- Times: this node publishes de times of the different sub-processes, such as the image preprocessing time, the time to process the image using the NN, the time to extract the depth value, etc.

Node execution:

When a RGB or a depth map are received from the ZED camera, two functions (*callback* functions) are executed separately.

When, a depth map is received, the depth callback function is executed. Inside this function, the ROS image is converted to an OpenCV image using the mentioned bridge. This depth map is stored in a variable.

Then, this variable is retrieved in the RGB image callback function when an RBG image is received from the ZED camera. This ROS image is also converted to an OpenCV image. Having the depth map and the RGB image, the deep learning algorithm can be executed, and then, the depth value extraction.

- Detection using deep learning and the RGB image:

First, the RGB image is pre-processed similarly as it was done with the training images. To do it, it is important to know the resolution of the image. As it was explained, two resolutions are available: 1080p and 720p. Depending on the resolution, the image is resized to a one quarter or to one a third of its size, respectively.

Then, this resized frame is cropped to the NN size (224x224). In this case, the central part of the image is cropped.

This cropped frame is converted to a PyTorch tensor and the batch size dimension is added. This is necessary because the network input is a tensor of dimensions (N, C, H, W). In this case, the batch dimension is set to 1, since there is only one image. Therefore, the input tensor has the dimensions (1, 3, 224, 224).

Then, this image tensor enters the network. As when validating the model, the forward propagation step is done and the output is obtained. As a remainder, the output is the heat map with the probabilities of a pixel containing the keypoint. The maximum value is extracted (probability), as well as the pixel coordinates (u,v) and the duration of this process. These values will be displayed together with the RGB image.

Then, this coordinates of the keypoint in the network image are transformed to the coordinates of the keypoint in the original frame. Considering the difference in image sizes and the operations that have been applied, this is a simple mathematical operation. In addition, it is important to remind that the (0,0) point is in the left-top of the image and here, the first coordinate corresponds to the number of row of the matrix and the second coordinate correspond to the number of column.

- Depth extraction using stereo vision and the depth map:

Once the coordinates of the keypoint have been obtained, they are used to extract the distance value. As previously explained, the (u,v) coordinates of the RGB image correspond to the (u,v) coordinates of the depth map. Therefore, to obtain the distance it is necessary to get the depth value of the pixel in the (u,v) coordinates.

The distance value is only extracted if the probability is greater than the threshold value. Otherwise, the distance value is set to “None”.

Finally, these results are displayed using OpenCV library. And this process is repeated each time a frame is received.

6.2.2. Results

In this section, the results obtained from the combination of deep learning and stereo vision techniques are shown. To clearly show the results in order to understand the different failures and successes that exist, a sequence of frames has been extracted from a video. In addition, in all these images, the different failures and successes that are observed are indicated.

The resolution in this video is **1920x1080**.

First, as long as no one appears in the video, the algorithm does not detect or mark anything (Figure 6-88). As previously seen, the probability is close to zero (0,02).

Figure 6-88: Video frame. Probability=0,02.

At the moment someone arrives, while the person is outside the area being analysed, the point remains undetected. For example, in Figure 6-89, the keypoint region is still cut by the edge of the 224x224 frame. The region that remains inside the rectangle will be the one that will pass through the network. So it is understandable that the algorithm still doesn't detect anything.

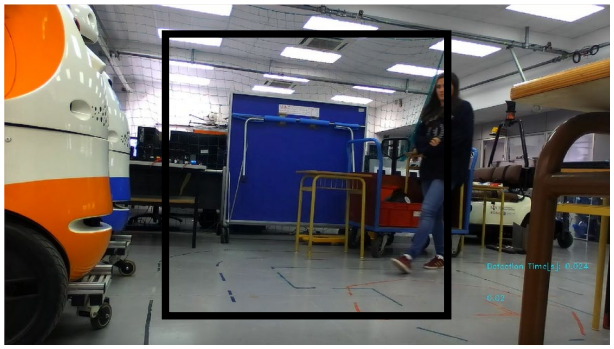


Figure 6-89: Video frame. Probability=0,02.

Then, the person enters the area where it can already be detected. Even so, as it was seen in section 6.1.6, sometimes the algorithm fails in detection, especially when the person is far away and when he is moving. This is the case in Figure 6-90, where the person's face is blurred and causes the algorithm not to detect the keypoint. However, the probability is higher (0,63) than in the previous frames, but it is still below the threshold, which is 0,7.

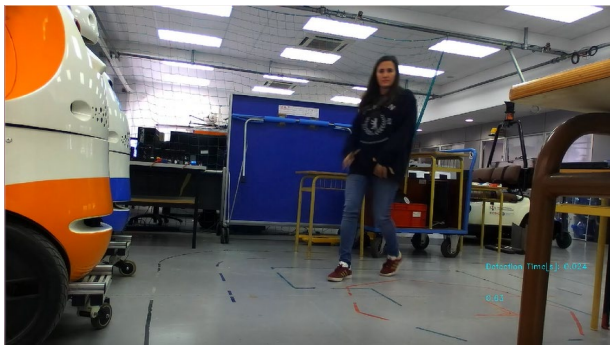


Figure 6-90: Video frame. Probability=0,63.

Finally, the algorithm detects the person (Figure 6-91). However, the problem that exists here is related to the distance value. The value of the distance is 9,21m, while the person is not at 9,21m. So the error here is that the distance shows a much larger value than it should.

Figure 6-91: Video frame. Probability=0,98 and distance=9,21m.

Then the deep learning algorithm continues to detect the person correctly (Figure 6-92). Perhaps the point is slightly offset from where it should be, but calculating the distance there is not a problem. However, here there is still a problem with distance.

In this case, the distance value is “NaN” (Not a Number), which is an acronym that is generally used in programming languages to express a result that is impossible to calculate due to, for example, an indeterminate form. When the ZED camera can’t determinate a value when computing the depth map, it puts a “NaN” value.

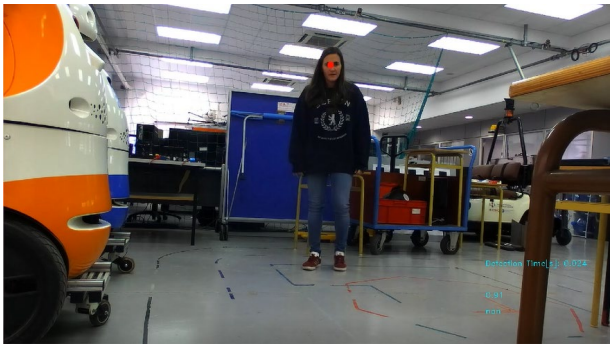


Figure 6-92: Video frame. Probability=0,91 and distance=nan

In the following frame (Figure 6-93), the distance takes a reasonable value (3,47m). As previously seen, the maximum distance that the deep learning algorithm can detect with the 1080p resolution is 4m. Therefore, at 3,47m, it can perfectly detect the keypoint. However, in this case, the keypoint is slightly shifted to the left. Even so, it is not a problem when calculating the distance, since that point continues being part of the person’s face.

Figure 6-93: Video frame. Probability= 0,79 and distance=3,47m.

In the next 3 frames, as the person is still far away and moving, the algorithm fails in detection. In the first, where the person is squatting and their face is blurred, the probability of detection is 0,16. In the following (which is the one shown in Figure 6-94) the person is still not detected. And in the third, where the person is standing up and his face is blurred, the probability is still very low (0,07).

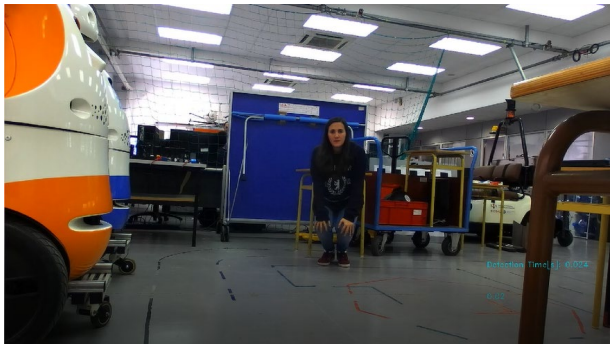


Figure 6-94: Video frame. Probability=0,02.

From here, as the person approaches, the detection improves. Nonetheless, the problem with distance ("NaN" values or much higher values than they should) remains unsolved.

In the next frame (Figure 6-95) the person is detected and located at a distance of 3,99m. Although the keypoint is shifted to the left, the distance value seems reasonable. However, in Figure 6-93, it seems that the person is in the same place but the distance is less in that case (3,47m).

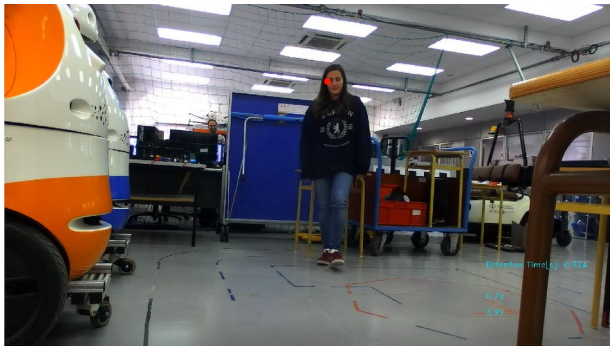


Figure 6-95: Video frame. Probability=0,79 and distance=3,99m.

Below are a series of frames that seem to consistently detect and locate the person. As the person gets closer, the distance decreases (Figure 6-96, Figure 6-97, Figure 6-98, Figure 6-99). In these images, although the person moves, by being closer, the algorithm manages to detect it. Also, the distance value has not increased dramatically even when the person squats.

Figure 6-96: Video frame. Probability=0,93 and distance=3,86m.

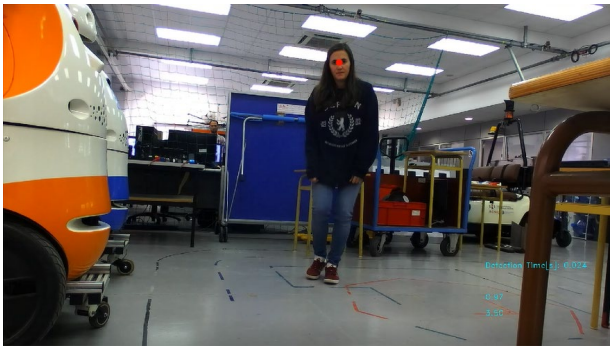


Figure 6-97: Video frame. Probability=0,97 and distance=3,5m.

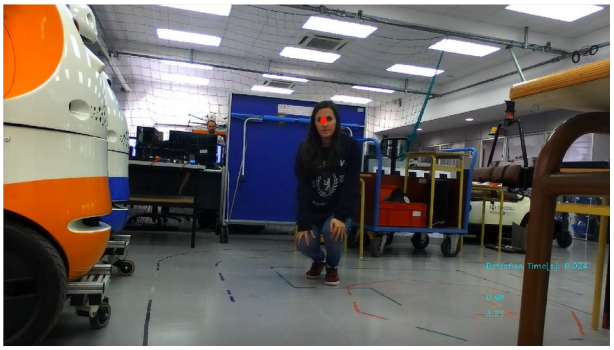


Figure 6-98: Video frame. Probability=0,98 and distance=3,23m.

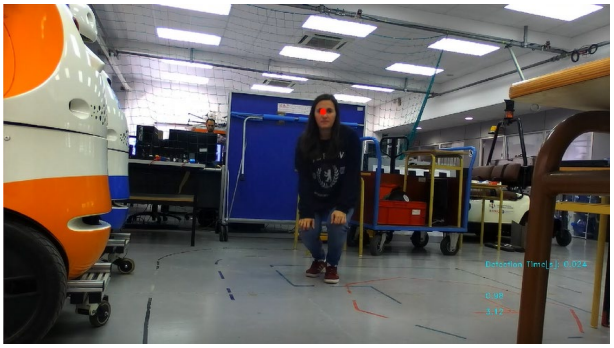


Figure 6-99: Video frame. Probability=0,98 and distance=3,12m.

Then, in the next two frames the distance problem appears. First (Figure 6-100), when the person stands up, the value of the distance is equal to infinity. The infinite value is given because the ZED camera, when calculating the depth map, gives all those points that are more than 10m, an infinite value. Therefore, in this frame, the algorithm is giving a distance of more than 10m, which is a serious error. And in the second frame (Figure 6-101), when the person moves laterally, the distance value is equal to 9,86m.

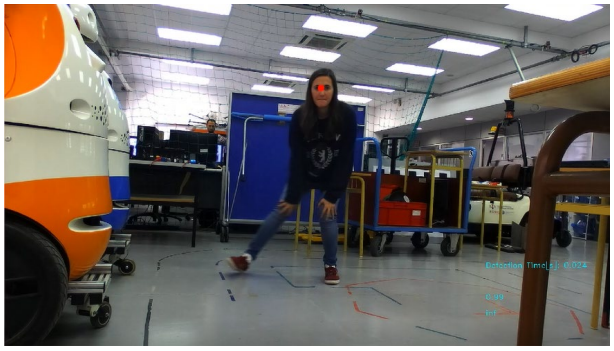


Figure 6-100: Video frame. Probability=0,99 and distance=inf (more than 10m).

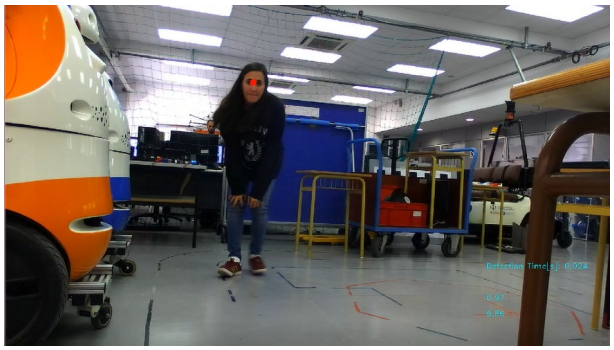


Figure 6-101: Video frame. Probability=0,97 and distance=9,86m.

When the person remains still in the same place, the distance takes a reasonable value of 2,81m (Figure 6-102). Furthermore, considering that the last correct value was a distance of 3,12m (Figure 6-99) and from there, the person has only advanced a little, this value also seems to be correct.

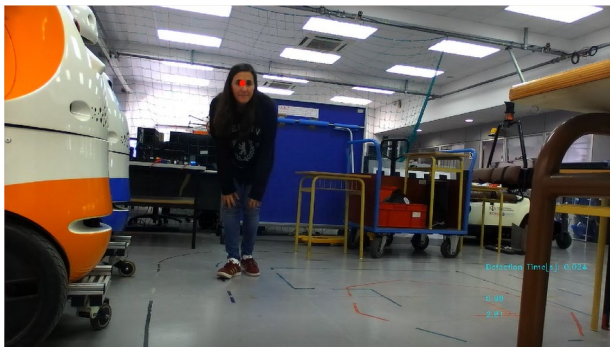


Figure 6-102: Video frame. Probability=0,98 and distance=2,81m.

The person then moves again (stands up and moves laterally) causing the distance value to increase dramatically. This time, it also needs two frames to position itself on the depth map. One of these two frames is shown below (Figure 6-103).

Figure 6-103: Video frame. Probability=0,76 and distance=7,81m.

Closely, the problems remain the same. The deep learning algorithm detects better when the person is close, although being close is easier to get out of the study area. For example, in Figure 6-107, the person's face is near the top edge of the image and does not fit inside the box. However, being closer, the point location is more precise, that is, the algorithm can place it in the correct position even if the image is blurred (Figure 6-104 and Figure 6-112).

Regarding the algorithm that extracts the distance, it gives the same errors as for farther distances. As the person moves to the side or when standing up, the distance value increases dramatically.

Next, the end of the frame sequence is shown, extracting the frames where there were distance errors to see how consistent this value is when the person approaches and remains still on the site. As shown in the figures from Figure 6-104 (2,34m) to Figure 6-114 (0,7m), the distance decreases as the person approaches.

When the person moves very little from one frame to another, the distance varies very little (Figure 6-105 (2,10m) and Figure 6-106 (2,04m)).

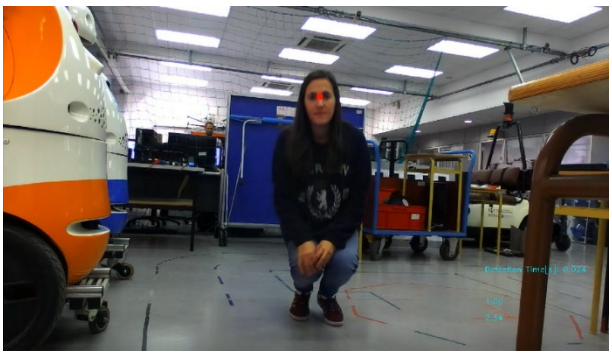


Figure 6-104: Video frame. Probability=1 and distance=2,34m.

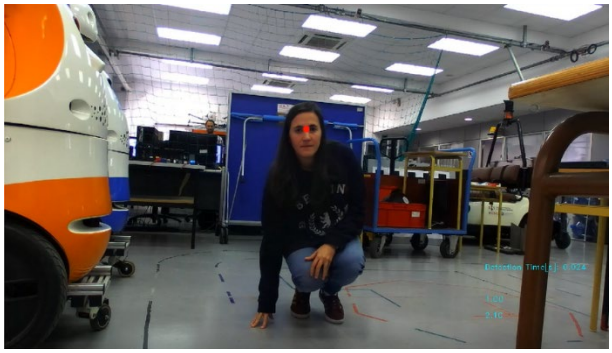


Figure 6-105: Video frame. Probability=1 and distance=2,10m.

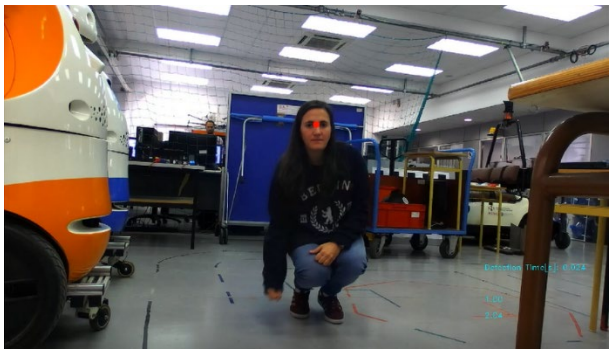


Figure 6-106: Video frame. Probability=1 and distance=2,04m.

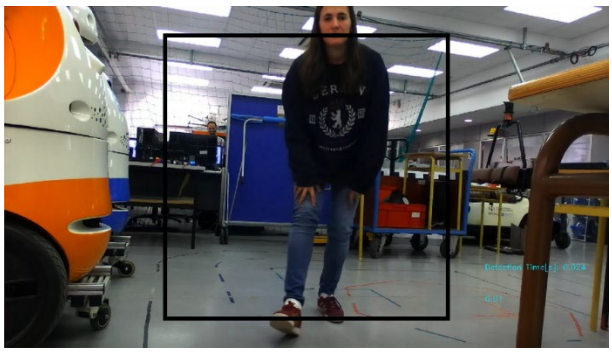


Figure 6-107: Video frame. Probability=0,01m

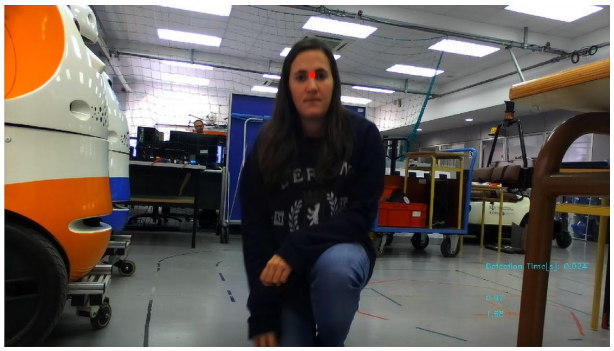


Figure 6-108: Video frame. Probability=0,97 and distance=1,68m.

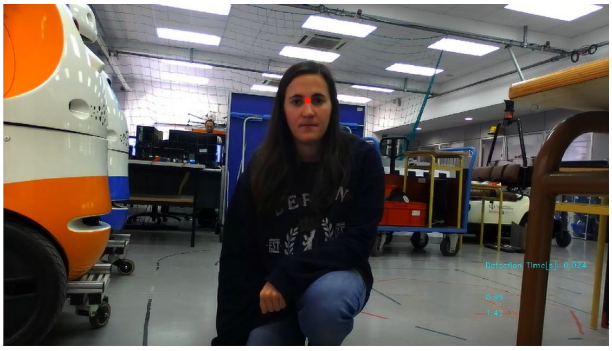


Figure 6-109: Video frame. Probability=0,99 and distance 1,42m.

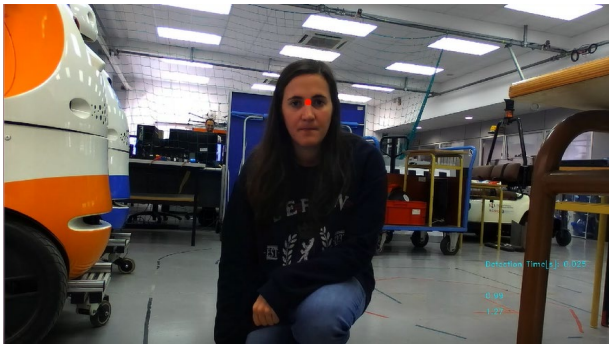


Figure 6-110: Video frame. Probability=0,99 and distance 1,27m.

Figure 6-111: Video frame. Probability=0,99 and distance 1,11m.

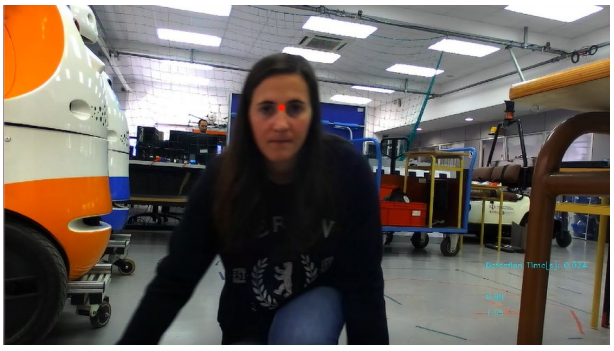


Figure 6-112: Video frame. Probability=0,99 and distance 1.05m.

Figure 6-113: Video frame. Probability=1 and distance=0,88m.

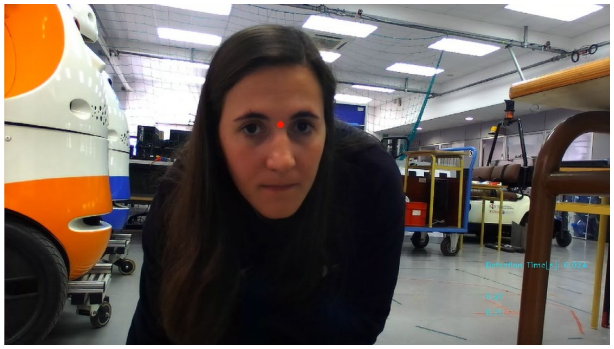


Figure 6-114: Video frame. Probability=0,97 and distance=0,70m.

In the last frame, the distance takes a value of minus infinity (Figure 6-115). This is due to the fact that when a distance is less than 0,7m, the ZED camera sets a –inf value. Therefore, the –inf value means that the person is closer to 0,7m.

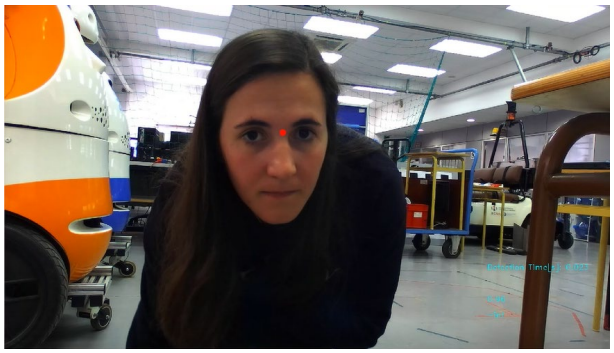


Figure 6-115: Video frame. Probability=0,96 and distance=-inf (less than 0,7m).

As seen in section 6.1.6, the deep learning algorithm works well with negative samples. When the frame is clearly a negative sample, the probability is very small (less than 0,1).

On the other hand, for positive samples, the algorithm works better at a closer distance, since it can detect people in almost all cases, even under poor conditions.

In this section, it is demonstrated that with the 1080p resolution the algorithm shows reasonable distances below 4m. When the distance takes a much higher value, it is assumed

that the algorithm is failing due to some problem in the code. In the following section, some of the errors seen in this section will be solved.

To verify at which distance, with 720p resolution, the algorithm can no longer detect the person, some videos with that resolution have been recorded too. It has been observed that, with a 720p resolution, the maximum distance where a person can be detected is 2,25m (Figure 6-116).

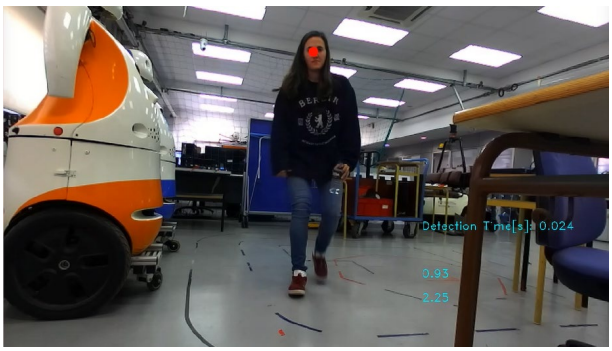


Figure 6-116: Video frame. Resolution=720p. Probability=0,93 and distance=2,25m.

To conclude, as seen in section 6.1.6, with 720p resolution less precise results are obtained, and the maximum distance where a person can be detected is less than if the 1080p resolution is used; in this section it has been verified.

In Annex 12.8, there are more examples of images with 720p and 1080p resolutions.

7. Discussion of results

7.1. Types of errors

7.1.1. Deep learning algorithm errors

- Far distances:

  As seen in section 6.2.2, with a 720p resolution, the maximum distance where it is possible to detect the person is 2,25m. And with a 1080p resolution, the maximum distance is 4m. Therefore, a person who is at a distance greater than these values, the algorithm will not be able to detect it.

- Blurred images:

  The algorithm sometimes fails when trying to detect people who are far away (for example, more than 3m with the 1080p resolution) when the person's face is blurred and the region is poorly defined. This normally happens when the person moves quickly. Fortunately, when the person is close, detection is better. If this were not the case and the person was not detected, there would be a serious problem.

- Shifted point:

  In some images where the person is more than 3m away (with the 1080p resolution), the keypoint appears slightly shifted (usually to the left).

- Region size:

  Sometimes when the person moves laterally or stands up, it leaves the cropped region. If the person's face is outside the region, it does not pass through the network and nothing is detected.

- Half-profile images:

  Sometimes half profile people are detected, since most of the keypoint region can be seen.

7.1.2. Stereo vision algorithm errors

- “NaN” values:

  At some points in the video, the distance takes an undefined value (“nan”). This does not happen at a specific time; it just happens when the ZED camera software can't calculate that depth value.

- Distance value dramatically increases:

  This is probably the most serious error in terms of distance. As seen in section 6.2.2, when the person moves laterally or stands up, the distance usually takes a much higher value than it should. Also, this happens both when the person is far (4m) and when the person is close (1m). Telling a robot that a person is much farther than they really are can cause serious accidents; the robot could move forward and collide with the person.

- ZED camera frame-rate:

  In general, when using real-time algorithms, the smallest resolution (VGA) is used. Still, this resolution did not work and we had to work with larger resolutions (720p and 1080p). When using higher resolutions, the images contain more pixels and consequently, the images are sent very slowly. For example, when the ZED takes the 1080p resolution images and sends them to the Jetson, it has to send 40 million bits per image, which is a huge cost of time.

  Looking at the frequency which the topics are received with the 1080p resolution, 1,294 frames per second are received, or what is the same, one frame is received every 0,77 seconds. With the 720p resolution, as the images are smaller, this improves; the speed is 4 FPS. Even so, knowing that a 720p resolution video contains 60 frames per second, 4 FPS is still a very slow speed.

  In this case, you will have to decide which is preferred: detect people at a greater distance and receive more spaced frames over time, or to detect people at a shorter distance but receive more frames per second.

  This will be the bottleneck of the project, since the deep learning algorithm with the distance extraction, is faster; it takes a maximum of 40ms to carry out all the operations. Therefore, 25 frames per second could be analysed. However, this bottleneck only allows analysis of up to 4 frames per second.

  For this reason, all the topics published by the algorithm (distance, resulting image and times), are published at a topic-per-second ratio equal to 1,294 (using 1080p resolution).

7.2. Algorithm improvements

- Patch:

  A patch has been applied to compute the distance value in order to solve the “NaN” values problem. The patch applied is a 5x5 size square, centered on the keypoint.

  Since the “NaN” values do not give any useful information, a path is applied to obtain more distance information. Also, it is more robust to extract the depth value using the

mean of a group of pixels instead of a single pixel. So the keypoint and surrounding pixels are used (Figure 7-1).

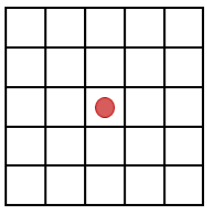


Figure 7-1: Patch shape

Using this patch and a NumPy function called np.nanmean, the mean of the pixel values in the patch is computed. If the result is a “nan” value, it will return “nan”. But this has never been observed because it is highly unlikely that the camera software does not know how to calculate the distance of any of the 25 pixels in the patch. Normally, the "nan" values are isolated points. Although there are many, it is unlikely to find many together in a region.

Different patch sizes have been tried (5x5, 6x6, 7x7) but they have all given reasonable distances. This is because the patch is still very small compared to the image size. Even so, the small size has been chosen for different reasons:

- Less calculations should be done (although they are not very time consuming).
- The results are still good and eliminates the problem of "NaN" values.
- In case the person is very far away, the patch must be as smaller as necessary for not taking a lot of face region nor taking values outside the face region.

- Distance filter:

In this case, a distance filter has been implemented. This is applied to solve serious problems if at some point the distance suddenly increases.

To do this, two types of filter have been tested.

The **first filter** consists of the following: knowing that the algorithm only detects people who are at a maximum distance of 4m, any distance value greater than 5m is taken as "None". That is, the value is assumed to be incorrect and is not sent to the robot. The algorithm will wait for the next frame to receive the distance information.

This first filter is very simple. Basically it consists of applying a threshold value to limit the distance. This filter works well with videos made, since when the distance value increases it only increases at very large values (greater than 5m). This means that it has never increased suddenly from 1m to 3m, for example. The reason for this will be explained in the next point.

The **second filter** consists of a filter that depends on the previous distance value. In this case, if the distance is greater or less than 2m with respect to the previous value, is considered that it is not correct and, therefore, null. This value of 2m has been chosen because it is highly unlikely that a person has moved two meters forward or backward, in just a single frame.

Therefore, instead of doing it with a fixed threshold, the previous value is used to determine if the new value is feasible or not.

In addition, this second filter prevents the distance from suddenly increasing, both far and near (for example, from 1m to 3m). Although the algorithm has to do more calculations and save the previous distance value, this does not take much more time (Table 7-1).

	Time to extract the depth value (ms)
First filter	1,356
Second filter	1,535

Table 7-1: Time to extract the depth value (ms) with the first and the second filter

Therefore, undoubtedly, the second filter is the best option to implement. It is a simple filter but it has demonstrated working well for this application. Even if the algorithm manages to calculate distances well, this filter will guarantee that there are no errors. Therefore, the filter is applied for security reasons.

- Callback functions:

  It is not a coincidence that the distances suddenly increase when a person moves.

  Using the timestamp function, the time at which the RGB image and the depth map are stored (in each callback function) has been checked. So it has been seen that there is a delay of 0,773 seconds between the depth map and the color image. This means that in the same operation, the new color image and the previous depth map are being used.

  In the current RGB image, the algorithm detects the keypoint at the coordinates (u, v) and when it searches for these coordinates on the previous depth map, it finds a point at the background of the image. For this reason, the distance suddenly takes large values, sometimes coinciding with the ceiling or the wall behind. Since there are no objects behind the person that are very close, the distance has never gone, for example, from 1m to 3m.

  For example, in Figure 7-2, when the person is detected in the current frame (u,v), the algorithm searches the distance value in the depth map but, for the previous frame. In this case, these coordinates (u,v) coincide with those of a point on the ceiling. This is why the distance takes such a large value (9,21m).

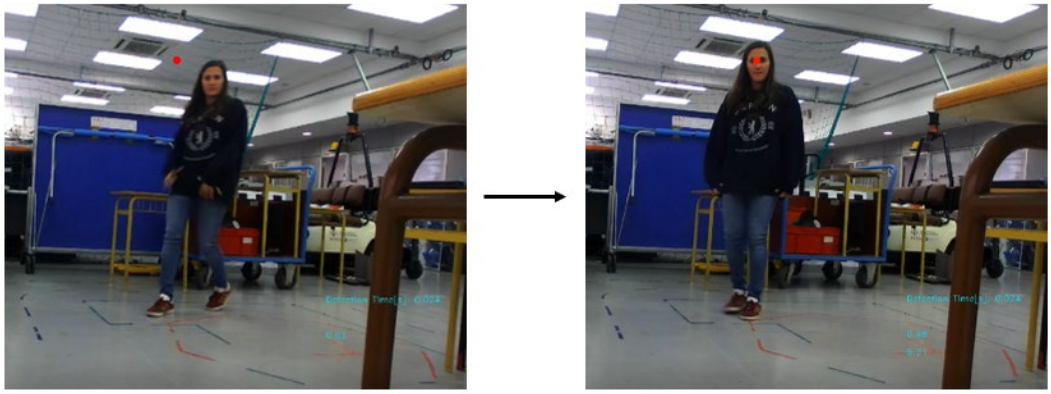


Figure 7-2: Video frames. On the left, the previous frame and on the right, the current frame.

By putting the code of the entire process inside one callback function, it blocks the other callback function. This means that while all the operations are performed within one callback function, the other cannot be executed.

So what has been done to try to eliminate this delay is to cause the entire process of deep learning and depth extraction to be carried out in the main program. Therefore, each callback function should only take the data from the RGB image and the depth map and save it in to two variables.

Therefore, verifying the times with timestamp, it has been seen that now there is no delay. Furthermore, more tests have been done and it has been observed that the distance value always takes a reasonable value.

Consequently, it can be deduced that the depth map and the distance extraction system are reliable. Even so, the applied improvements related to the distance filter have also been implemented for security. Furthermore, it can be seen here that the first distance filter applied does not make sense knowing these reasons and that it is better to apply the second one.

All these improvements can be seen in the code in Annex 12.7.2 and their results can be seen in section 7.4.

7.3. Unsolved problems

7.3.1. Deep learning algorithm problems

- Far distances:

  This problem is due to the images with which the network has been trained. Since many of the images are of faces and some more contain people up close, there are very few images containing people more than 4m away.

Also, the more the image is reduced, the more difficult it is to detect people who are far away. For example, since 720p resolution images end up shrinking more than 1080p resolution images, with the first, the person can be detected up to 2,25m and with the second, it can be detected up to 4m.

In this case, you would have to decide the resolution you want and how much to reduce the image before cropping it.

- Blurred images:

  This problem could not be solved, as images are sometimes blurred if the person moves abruptly. When the person or camera moves slowly, the image may be blurred, but if the person is not far away (for example, less than 3m with 1080p resolution images), the algorithm will be able to detect it with sufficient precision. If the person is no farther than 3m and the keypoint can be minimally identified (i.e. a human would know where to locate it), the algorithm, in most cases, also detects it.

  Accordingly, this error (false negative) generally occurs when the person is far away. Therefore, this would not be a serious problem since the robot could stop and wait to detect the person in the next frame. Also, it is quite likely that the algorithm will be able to detect the person in the next frame, as the person usually stands still at some point.

- Shifted point:

  This problem occurs in images where the person is at a far distance. Even so, it is a problem that does not worry too much, since, as has been seen, the detected point is still a point on the face. Therefore, the distances will be very similar. In addition, in the next frame, it generally detects it more accurately.

  In no case has the point been seen to come out of the person's face and indicate a point on the image background. Otherwise this would be a problem and should be solved. Still, the deep learning algorithm is highly unlikely to detect the keypoint at a point outside the person's face. To solve that, the network has been trained with negative images and positive images. Typically, as seen, the algorithm will either fail to detect people in profile or not detect people when it should. But it will not place a point in any other region than the face one.

  Region size:

  The person leaves the study area or not, depending on the resolution of the images and how much the image has been downscaled before cropping it to pass it through the network. Even so, since the ZED image is rectangular and the image that passes through the network is square, information on the sides will be lost.

To avoid this, what can be done is to implement a Visual Servoing system to control the pan and tilt of the camera in order to move the camera according to the movement of the person's face.

To solve deep learning detection problems, we have tried to implement a probability filter.

As explained, when the algorithm detects the person well, the probabilities are quite high; from approximately a probability of 0,79 when the person is far away, to a probability of 1 when the person is close.

On the other hand, when the person's face does not appear in the image, the algorithm gives very low probabilities (less than 0,1).

So, to make it more conservative, the threshold probability has been set to 0,7. But then, what has been observed is that in some cases where the person is far away and moving (Figure 6-90), the probability is slightly below the threshold and therefore, the algorithm deduces that there is no people's face there.

At first, it was thought that this could be solved, for example, by applying a filter. As seen in the code below, the algorithm checked if the person had been detected in the previous frame, and if that was the case, a lower threshold value (0,6) was used. That is, if the person had been detected in the previous frame (probability> = 0,7), then if in the current frame the probability was 0,6, this probability was "corrected" to 0,7 so that the algorithm would mark the person and take his distance.

```
...
crop_frame, frame_tensor = self._preprocess_frame(frame, is_bgr)
hm, uv_max, prob, elapsed_time = self._detect_person(frame_tensor)
uv_max = self._recover_uv_orig(uv_max)
final_prob = self._filter_prob(prob) if self._do_filter_prob else prob
if self._do_filter_prob:
self._update_prob_filter(prob)
....
def _filter_prob(self, prob):
        if self._detected_in_previous_frame:
                return 0.7 if prob >= 0.6 else prob
        else:
                return prob
def _update_prob_filter(self, prob):
        self._detected_in_previous_frame = prob >= self._prob_treshold
```

Even so, for this project it has finally been decided not to apply this modification. The reason is that when trying to solve a problem, another one that could be more serious appears. When testing it, it was seen that when the person was in a half profile, sometimes it was not detected because the probabilities were 0,5 or 0,6. By adding this filter, it is easier to detect people in profile (Figure 7-3).

Figure 7-3: Video frame. Resolution=1080p. Probability=0,61.

In this case, it is important to decide which is preferred. This decision will depend on how the behaviour of the robot is programmed. In this project, the following has been assumed: when the deep learning algorithm receives nothing, the robot stops, so a false negative would not be a serious problem. On the other hand, if this modification is used and the algorithm detects a person in profile (false positive), it will approach it, and if it is a person who is crossing in front, the robot could collide with it. Therefore, it has been decided that false negatives are less serious than false positives.

Although the algorithm still continues to detect some people slightly in profile because the probabilities are very high, by not using this modification we avoid detecting many more profiles that would not be detected under normal conditions.

7.3.2. Stereo Vision algorithm problems

- ZED camera frame-rate:

  This problem remains unsolved, as the smallest resolution does not work. Still, it has been possible to verify that the algorithm developed in this project works, both to detect people and to know at what distance they are.

  A possible solution to this problem is to use another stereo camera like the Real Sense that compresses the images before sending them. The ZED camera works with uncompressed images, making it costly to send such large images

7.4. Improved results

People between 0,7m and 4m away will be detected using a 1080p resolution. Between each frame there will be approximately 0,77 seconds due to the bottleneck of the ratio of topics per second received from the ZED camera. With a 720p resolution, the ratio improves, but the maximum distance to detect is less than with 1080p resolution.

Using the patch to calculate the distance avoids the "NaN" values (Figure 7-6). In order to see the difference between the result with a patch and the result taking only the distance from the keypoint, the two values have been shown in the image. As you can see, there is not much difference between taking the average distance of all the points of the patch and that of the point, but there is a difference when the distance value at that point is "NaN".

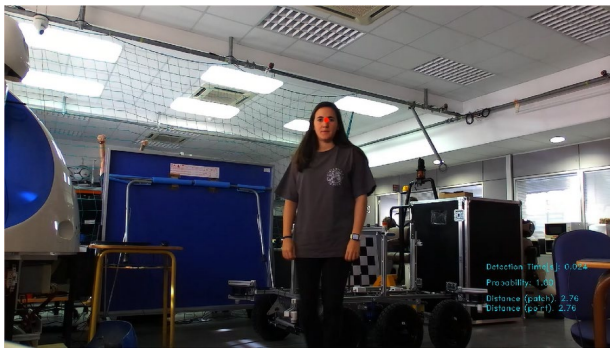


Figure 7-4: Video frame. Resolution=1080p. Probability=1,00, distance using the patch=2,76m and distance using the keypoint=2,76m. There is no difference.

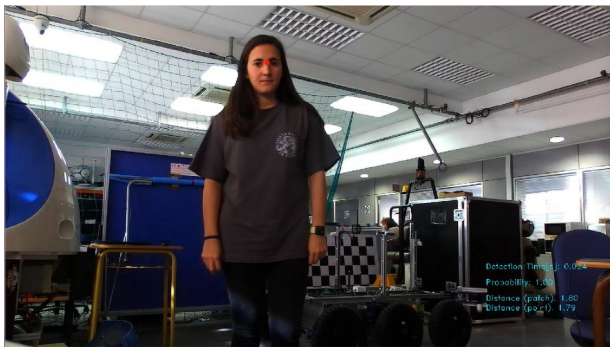


Figure 7-5: Video frame. Resolution=1080p. Probability=1,00, distance using the patch=1,80m and distance using the keypoint=1,79m. There is a little difference.

Figure 7-6: Video frame. Resolution=1080p. Probability=1,00, distance using the patch=2,35m and distance using the keypoint=nan. There is a difference.

The reason why there is not much difference between these two values is that, as seen in Figure 7-7, looking at the depth map, all points on the face are at the same depth.

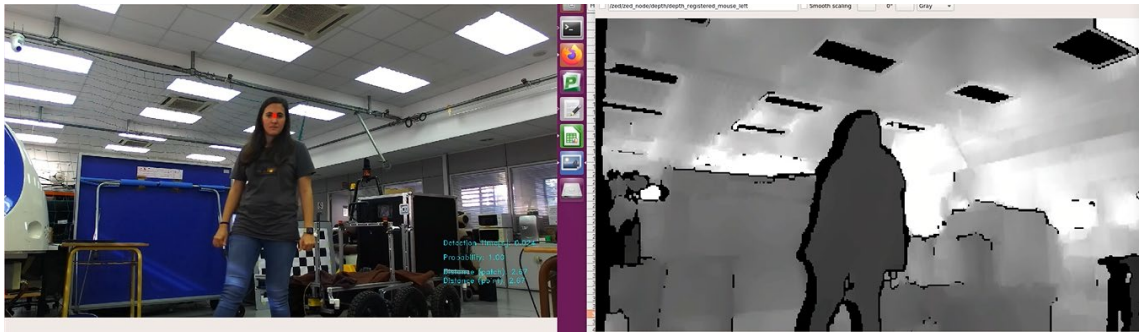


Figure 7-7: On the right, the RGB image and on the left, the depth map.

On the other hand, by modifying the code to avoid the delay between the topics due to the saturation of the callback functions, and by using a distance filter, the distance has been achieved to take reasonable values. In order to see the change, an image is shown where the person has moved laterally and the distance is still reasonable (Figure 7-8 and Figure 7-9).

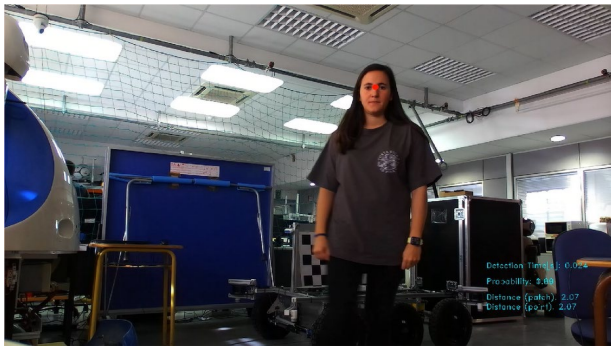


Figure 7-8: Video frame. Probability=0,99 and distance=2,07m.

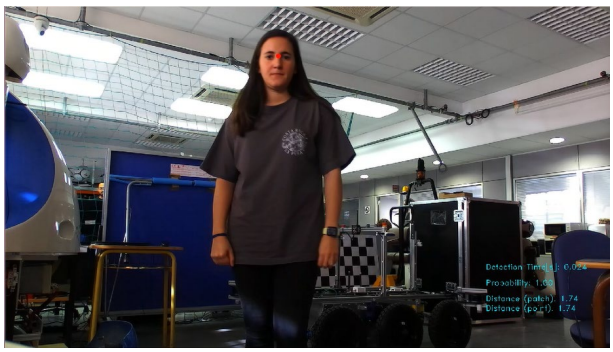


Figure 7-9: Video frame. Probability=1,00 and distance=1,74m.

In addition, it can be seen that when the person advances, the distance decreases and when the person moves away, the distance increases.

To verify that the results are consistent, the results obtained from a video with 25 processed frames have been plotted (Figure 7-10). These images are shown in Annex 12.10. As seen in the video frames, the person first approaches, then moves away from the camera and finally, approaches again.

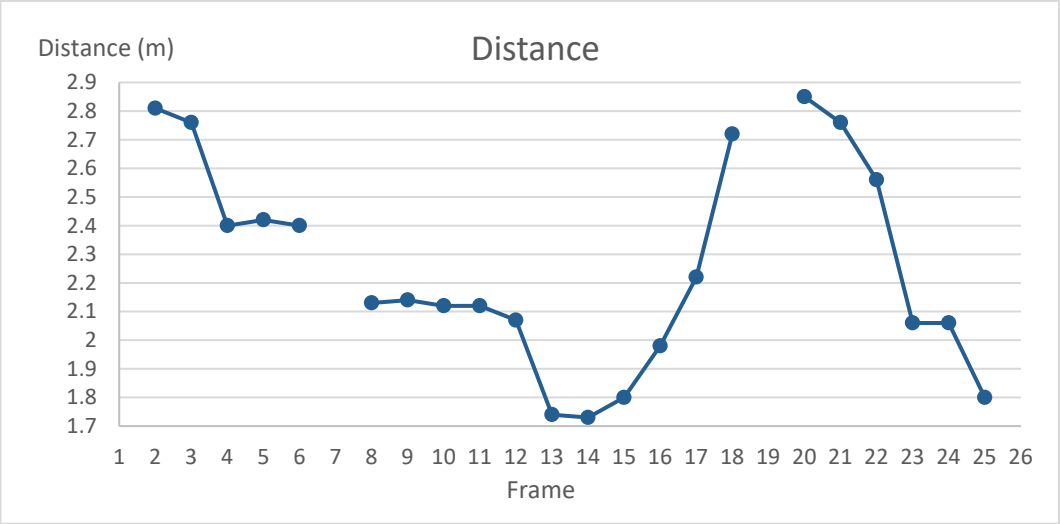


Figure 7-10: Distance value in each video frame.

As Figure 7-10 shows, the distance values are continuous and reasonable according to the image sequence. However, there are three discontinuities. These are due to:

- Frame 1: the image is blurred and the person is almost 3m away.
- Frame 7: the image is blurred because the person is moving.
- Frame 19: the person is in profile. It is a negative sample, so the result is correct.

Therefore, the person also stands still in the same place for some time to also see if the distance maintains the same value (or almost the same value). For example, from the 8th frame to the 11th frame.

The distances range from 1,7m to 2,9m approximately. A small range (1,2m) has been taken to see that the algorithm locates the person fairly accurately. In addition, a middle distance has been taken, neither too far nor too close so that the errors are not detection errors. Annex 12.9 shows images with distances less than 1,7m and more than 2,9m to see that the algorithm also works with these distances and also takes reasonable values.

Since in this case there is no ground truth distances, it is difficult to know exactly how much error there is. For this reason, you can only see that the data are continuous and goes according to the situation. For this reason, we can conclude that these values are consistent.

In the x-axis there is the frame number. As it is known that the frame rate is 1,294 FPS due to the ZED camera delay, this axis can also be represented by the time. However, the frame number has been chosen to make it easy to see the distance value with its corresponding image in the Annex 12.10.

Finally, what the object detector was already doing well, continues doing well when combined with stereo vision, since that part remains unchanged. For example, the algorithm gives very

small probabilities when there is no person in the image or if it is seen from behind or is completely in profile (Figure 7-11).

Figure 7-11: Video frame. Probability=0,06. Profile view.

In addition, it detects people from near and far (Figure 7-12), even when images are blurred. Even so, as discussed, in cases where the person is far away and moving, it is more difficult for the algorithm to success.

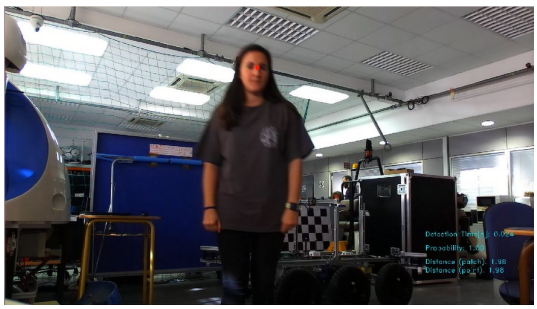


Figure 7-12: Video frame. Probability= 1. Distance=1,98m. Blurred image.

To verify the object detector accuracy, the same 25 frames have been used (Figure 7-13). In this case, the error detection is computed in terms of distance between pixels. To do this operation, the distance between two points is computed using the ground truth coordinates and the keypoint coordinates.

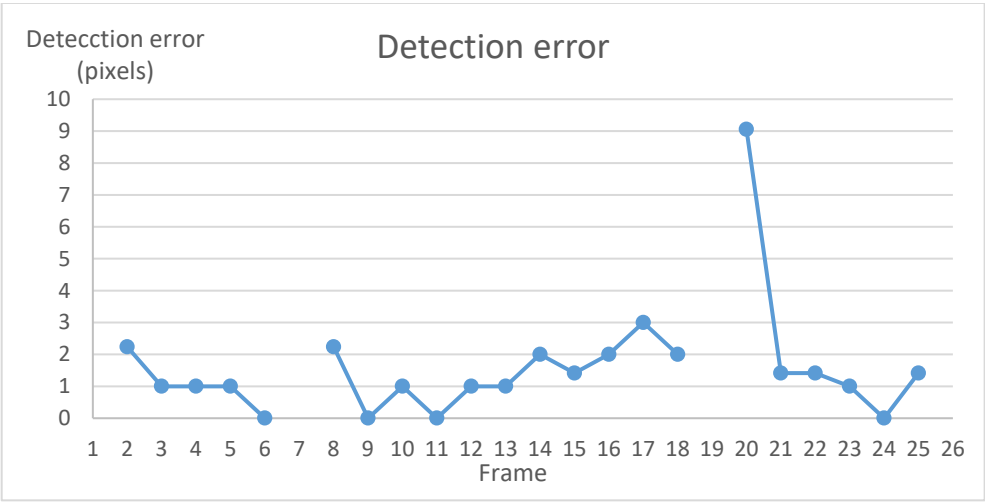


Figure 7-13: Detection error (pixels) in each video frame.

Note that the same three frames that could not be detected have no error value. However, two of them (the 1st and the 7th) contain the keypoint but the algorithm does not detect it, since the images are blurred. In these cases, the error takes a larger value than those on the graph. On the other hand, the other frame (19th) is a negative sample and is no detected, so there is no error.

As can be seen, in most cases the error is very small (below 3). There is one point where the error is higher than the others (Frame 20). Figure 7-14 shows this frame and, as can be observed, the keypoint is slightly shifted to the left.

Figure 7-14: Video frame. Probability=0,9 and distance=2,85m.

To conclude, the errors are very small; less than 3 pixels within a 1920x1080 image. Even in the worst case, the error is still very small. That means that the keypoint always corresponds to a face point, which is good for calculating distance. Finally, the largest detection errors will be found in images where the person cannot be detected.

As previously stated, the processing frame time, is around 40ms. In the next table (Table 7-2), there are the times that correspond to each operation within the process using a 1080p resolution:

Times (ms)	No person	Person
Time to save the images	3,12	3,27
Time to preprocess the images	6,26	7,43
Time in the network (forward pass)	23,82	23,80
Time to recover (u,v)	1,31	1,31
Time to extract the depth value	0,30	1,54
Time to prepare and display image	1,79	1,72
Total time	**36,60**	**39,07**

Table 7-2: Process times (ms)

As it was supposed, where it takes more time is to calculate the heat map using the network. This time is almost 24ms and corresponds to the time shown in the figures. It is the time it takes for a 224x224 image to go through the network, until obtaining the heat map and the coordinates of the keypoint.

Then, being a quarter of the previous time, there is the image preprocessing time. This operation consists of resizing the image and cropping it. Because of the images are very large, these operations have to work with many data.

The other times are:

- Time to save the images: this operation consists of taking the data from the ZED topics, convert them to OpenCV images and store them in to two variables.
- Time to recover (u,v): this operation consists of computing the coordinates of the keypoint in the 1920x1080 image from the keypoint coordinates in the network image (224x224).
- Time to extract the depth value: time to obtain the distance value using the distance filter. In this case, there is a difference between negative and positive samples, since the distance is only computed on the negative images. Even so, there is not much difference since the operations are not time consuming.
- Time to prepare and display the image: this operation consists of taking the original image, draw on it the keypoint coordinates and the time, probability and distance value, and finally, publish this image.

As a conclusion, the time the algorithm takes to process a frame is very suitable to work in real time. The algorithm would be able to process 25 FPS but, due to the ZED delay, this could not be seen this project.

8. Budget

A budget is always needed for a research project.

The proposed budget for this project is presented in Table 8-1.

Concept	**Cost**			
A. Personnel Services	**Number of months [months]**	**Month rate [€/month]**		**Total cost [€]**
Graduate Researcher	12	500		6000,00
B. Equipment	**Total price [€]**	**Amortization [years]**	**Project duration [years]**	**Total cost [€]**
1 Personal Computer	700	5	1	140,00
1 Jetson AGX Xavier	712,37	5	1	142,48
1 ZED camera	349	5	1	69,80
C. Other	**Number of months [months]**	**Month rate [€/month]**		**Total cost [€]**
Formation courses	3	40		120,00
				6.472,28

Table 8-1: Budget of the project

The budget of the project is divided in three groups:

- the personnel services
- the equipment
- others

The cost of the researcher is corresponding to a UPC grant for students that are for working 20 hours/week on a research project.

At least, one personal computer is required to develop the network code.

Additionally, a Jetson AGX Xavier and a ZED camera are required to implement the real-time deep learning algorithm. Their prices correspond to the official websites of NVIDIA and Stereolabs, respectively.

Equipment costs are assumed to cover a 5-year amortization for each product. As the duration of this project is approximately one year, the cost of the equipment is the cost corresponding to one year of use.

Finally, the other costs are those that correspond to taking Coursera formation courses. Coursera has a fixed price of 40€ per month.

9. Environmental impact

As the realisation of this project has only consisted of programming an algorithm for object detection, it does not have much environmental impact. Perhaps, the training process is the one that could minimally affect the environment because it carries out in the NVIDIA Jetson AGX Xavier and requires electricity supply for a considerable number of hours.

Therefore, this section calculates the CO_2 emissions from training the deep learning algorithm. All the energy consumed in this process comes from electricity consumption.

As mentioned in section 5.4, Jetson AGX Xavier can consume up to 30W. To calculate the CO_2 emissions, it will be assumed that it consumes 30W.

This environmental impact study assumes that network training takes 24 hours to complete. Approximately the average duration of all the trainings has been taken.

Therefore, the total consumption of the training process is 0,72 kWh (Eq. 9-1).

$$Total\ consumption\ (1\ training) = \frac{30W}{1000\frac{W}{kW}} \cdot 24h = 0{,}72\ kWh \qquad \textit{Eq. 9-1}$$

Then, knowing that Spain's CO2 emission intensity is 0,241 kg CO_2/ kWh [49], $CO_{2(eq)}$ emissions due to the generation of electricity in one training are 0,207 kg $CO_{2(eq)}$ (Eq. 9-2):

$$CO_{2(eq)}\ emissions\ (1\ training) = 0{,}72kWh \cdot 0{,}241\frac{kg\ CO_{2(eq)}}{kWh} = 0{,}174\ kg\ CO_{2(eq)} \qquad \textit{Eq. 9-2}$$

Then, as in this project the model has been trained 8 times, using the Eq. 9-3 and Eq. 9-4, the total consumption is $5{,}76\ kWh$ and the total CO_2 emissions are $1{,}66\ kg\ CO_{2(eq)}$.

$$Total\ consumption = 0{,}72\ \frac{kWh}{training} \cdot 8\ trainings = \mathbf{5{,}76\ kWh} \qquad \textit{Eq. 9-3}$$

$$CO_{2(eq)}\ emissions = 0{,}174\ \frac{kg\ CO_{2(eq)}}{training} \cdot 8\ trainings = \mathbf{1{,}39\ kg\ CO_{2(eq)}} \qquad \textit{Eq. 9-4}$$

Finally, considering that a person consumes 38kWh in a month and that these emissions have been spread over 2 months, it can be concluded that these total emissions are not significant and can be neglected. Therefore, the total consumption of 5,76kWh can be included within the consumption of a person in three months, and all its emissions can be neglected.

As expected, the carbon footprint of this project does not have a great environmental impact.

10. Conclusions

Accurate and real-time results have been obtained in this project. Within the range of distances where it is possible to detect the person, the algorithm is able to accurately detect them and extract a reasonable distance value. Furthermore, the algorithm is capable of doing that at 25 FPS, an enough speed to work in real time.

Despite achieving the objective set at the beginning of the project regarding the person detection at a certain distance, a limitation has been found; people farther than 4m could not be detected. To minimize this limitation, it is very important to prepare the dataset to obtain the desired results. A representative sample of images of the parameters to be detected must be collected. They must work with an equal sample of images of the parameters to be studied.

Another factor to consider to minimize the limitations found in people's detection, is that images taken with the same camera that the robot will have implemented should be included (as a fine-tuning measure). In this project, the images have been reduced to a very small size to be able to pass through the network; this has also influenced in the detection of a person who is in a far distance. If the network was trained with larger images, these would not have to be reduced as much during preprocessing and this would aid detection.

There are some errors that, depending on the final application, become more or less important. In this project, it has been seen that the people's profiles are poorly defined causing confusion in their detection. In this case, as mentioned above, expanding the dataset with images of these characteristics and establishing detection criteria, this could be corrected. In any case, for this project, this error was not decisive in achieving the objective.

Combining deep learning techniques with stereo vision is an iterative process of idea-experiment-modifications; the first ideas are tested to see which results are obtained, then errors are detected, improvements are implemented, and again the model is tested and new results are seen. And this process is repeated successively. Another limitation of this project has been the time, so this iterative cycle has been carried out only by analyzing certain options, and during a specific period of time. Testing the model in situations with a different environment than the one studied, for example, outdoor or with different people, would be interesting to obtain more representative results.

The evolution of this project would be to modify the code in order to detect more people in the image and then develop a tracking algorithm to be able to follow the desired person.

This combination, the deep learning techniques with stereo vision, can have many applications today; for example, object handing tasks. By creating a new dataset for objects and following the same methodology, this could be done.

As seen in this project, deep learning offers multiple techniques for use in different applications. The Deep Learning community is in a continued growth, allowing its constant evolution, and its application in different areas.

This project is a small sample of the endless possibilities offered by deep learning.

11. Acknowledgements

I would like to especially thank my tutor Alberto Sanfeliu for giving me the opportunity to introduce myself in the research field, especially in the Artificial Intelligence field.

Thanks to Javier Laplaza for sharing with me his knowledge in deep learning.

Thanks to José Enrique Dominguez for helping me with technical issues.

References

[1] ZHOU, X; WANG, D; KRÄHENBÜHL, P. *Objects as Points*. 2019

[2] DO NHU, TAI; KIM, S.H.; YANG, HYUNG-JEONG; LEE, GUEE-SANG; NA, IN. *Face Tracking with Convolutional Neural Network Heat-Map*. 2018

[3] TEHNOKV. *Adapting a semantic segmentation pipeline for object detection.* July, 2018. [https://tehnokv.com/posts/segmentation-for-object-detection/, 4th April, 2020]

[4] E. GUERRA, A. PUMAROLA, A. GRAU AND A. SANFELIU. *Perception for detection and grasping. In Aerial Robotic Manipulation*. Vol 129 of Springer Tracts in Advanced Robotics, 275-283. Springer, 2019

[5] PUMAROLA, A. *Crawler Detector*. June, 2019 [https://github.com/albertpumarola/CrawlerDetector, 3rd October, 2019]

[6] RONNEBERGER, O; FISCHER, P; BROX, T. *U-Net: Convolutional Networks for Biomedical Image Segmentation.* 18th May, 2015.

[7] BUILT IN. *Artificial intelligence.* Chicago, 2019. [https://builtin.com/artificial-intelligence, 15th February 2020].

[8] MASOLO, C. *Supervised, unsupervised and deep learning. 7th* May, 2017. [https://towardsdatascience.com/supervised-unsupervised-and-deep-learning-aa61a0e5471c, 15th February 2020].

[9] COURSERA. *Deep Learning Specialization Course. Master Deep Learning, and Break into AI.* [https://www.coursera.org/specializations/deep-learning, July 2019]

[10] CHADDHA, Y. *Building a logistic regression using neural networks: cat vs non-cat image classification.* August, 2019. [https://medium.com/@yash.chaddha1997/building-a-logistic-regression-using-neural-networks-cat-vs-non-cat-image-classification-d8675208da5a, 20th December 2019]

[11] SHARMA, S. *Activation functions in neural networks*. 6th September, 2017. [https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6, 20th December 2019]

[12] DABBURA, I. *Gradient descent algorithm and its variants.* 21th December, 2017. [https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3, 21th December 2019]

[13] STANFORD. Course: *Convolutional Neural Networks for Visual Recognition*. Spring 2019. [http://cs231n.github.io/neural-networks-1/ and http://cs231n.github.io/neural-networks-2/, 21th December 2019]

[14] KATANFOROOSH, K. KUNIN, D. *Initializing neural networks.* 2019. [https://www.deeplearning.ai/ai-notes/initialization/, 21th December 2019]

[15] BROUNLEE, J. *Why initialize a neural network with random weights?* 1st August, 2018. [https://machinelearningmastery.com/why-initialize-a-neural-network-with-random-weights/, 21th December 2019]

[16] SHAFKAT, I. *Intuitively Understanding Convolutions for Deep Learning.* 1st June, 2018. [https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1, 27th December, 2019]

[17] BROWNLEE, J. *A Gentle Introduction to Pooling Layers for Convolutional Neural Networks.* 22nd April, 2019. [https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/, 10th January, 2020]

[18] MISSINGLINK. AI. *Fully Connected Layers in Convolutional Neural Networks: The Complete Guide.* [https://missinglink.ai/guides/convolutional-neural-networks/fully-connected-layers-convolutional-neural-networks-complete-guide/, 12th January, 2020]

[19] PYTORCH. *PyTorch documentation. PyTorch tutorials.* [https://pytorch.org/, 15th November, 2019]

[20] KHARKOVYNA, O. *Top 10 Best Deep Learning Frameworks in 2019.* 3rd June, 2019. [https://towardsdatascience.com/top-10-best-deep-learning-frameworks-in-2019-5ccb90ea6de, 26th March, 2020]

[21] BERENBAUM, D. *Take a deeper look at your PyTorch model with the new TensorBoard built-in.* 25th August, 2019. [https://towardsdatascience.com/https-medium-com-dinber19-take-a-deeper-look-at-your-pytorch-model-with-the-new-tensorboard-built-in-513969cf6a72, 26th March, 2020]

[22] TENSORFLOW. *TensorBoard: TensorFlow's visualization toolkit.* *[*https://www.tensorflow.org/tensorboard/, 26th March, 2020]

[23] DUBOVIKOV, K. *PyTorch vs TensorFlow — spotting the difference.* 20th June, 2017. [https://towardsdatascience.com/pytorch-vs-tensorflow-spotting-the-difference-25c75777377b, 26th March, 2020]

[24] HOLLÄNDER, B. *Logging in TensorBoard with PyTorch.* 4th September, 2018. [https://becominghuman.ai/logging-in-tensorboard-with-pytorch-or-any-other-library-c549163dee9e, 26th March, 2020]

[25] ROS. [https://www.ros.org/, 26th March, 2020]

[26] WIKI ROS. [http://wiki.ros.org/,26th March, 2020]

[27] TELLEZ, R. *What is ROS?* 20th September, 2019. [https://www.theconstructsim.com/what-is-ros/, 26th March, 2020]

[28] ROBOTICS BACK-END. *What is ROS*? [https://roboticsbackend.com/what-is-ros/, 26th March, 2020]

[29] QUIGLEY, M; CONLEY, K; GERKEY, B; FAUST, J; FOOTE, T; LEIBS, J; WHEELER, R; NG, A*. ROS: an open-source Robot Operating System. ICRA Workshop on Open Source Software. 3*. 2009.

[30] NVIDIA. *Autonomous machines*. [https://developer.nvidia.com/embedded-computing, 26th March, 2020]

[31] NVIDIA. *Jetson AGX Xavier* [https://www.nvidia.com/es-es/autonomous-machines/embedded-systems/jetson-agx-xavier/, 26th March, 2020]

[32] KAPERNIKOV. *Visual Odometry with the zed stereo camera.* [https://kapernikov.com/visual-odometry-with-the-zed-stereo-camera/, 26th March, 2020]

[33] STEREOLABS. *ZED Stereo Cameras Web Site*. 2017. [https://www.stereolabs.com/zed/, 26th March, 2020]

[34] ORTIZ, L; CABRERA, E; GONÇALVES, L. *Depth Data Error Modeling of the ZED 3D Vision Sensor from Stereolabs. Electronic Letters on Computer Vision and Image Analysis.* 2018.

[35] SHAH, T. *About Train, Validation and Test Sets in Machine Learning. 6th December, 2017. [*https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7, 17th October, 2019]

[36] NG. ANDREW; KATANFOROOSH, K. Deep Learning. Stanford. 2018. [https://cs230.stanford.edu/section/1/, 17th October, 2019]

[37] UNIVERSITY OF MASSACHUSETTS. *Labeled Faces in the Wild.* Massachusetts, 2018. [http://vis-www.cs.umass.edu/lfw/, October 29th, 2019]

[38] S. SENGUPTA, J.C. CHENG, C.D. CASTILLO, V.M. PATEL, R. CHELLAPPA, D.W. JACOBS. *Frontal to Profile Face Verification in the Wild.* IEEE Conference on Applications of Computer Vision, 2016. [http://www.cfpw.io/, October 29th, 2019]

[39] TSUNG-YI LIN, MICHAEL MAIRE, SERGE BELONGIE, LUBOMIR BOURDEV, ROSS GIRSHICK, JAMES HAYS, PIETRO PERONA, DEVA RAMANAN, C. LAWRENCE ZITNICK, PIOTR DOLLÁR. *Microsoft COCO: Common Objects in Context*. USA, 2015. [http://cocodataset.org/, October 29th, 2019]

[40] JIANPING SHI, RENJIE LIAO, JIAYA JIA. *CoDeL: A Human Co-detection and Labelling Framework.* IEEE International Conference on Computer Vision (ICCV), 2013. [http://shijianping.me/codel/index.html, October 29th, 2019]

[41] UNIVERSITY OF OXFORD. *VGG Human Pose Estimation datasets*. Oxford. [https://www.robots.ox.ac.uk/~vgg/data/pose/, October 29th, 2019]

[42] J. TIGHE, S. LAZEBNIK. *SuperParsing: Scalable Nonparametric Image Parsing with Superpixels.* European Conference on Computer Vision, 2010. [http://www.cs.unc.edu/~jtighe/Papers/ECCV10/, October 29th, 2019]

[43] N. DALAL, B. TRIGGS. *Histograms of oriented gradients for human detection*. IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2005). [http://pascal.inrialpes.fr/data/human, October 29th, 2019]

[44] TZUTALIN. *LabelImg.* Git code, 2015. [https://github.com/tzutalin/labelImg, November 11th, 2019]

[45] FILEINFO. *PKL File Extension*, 2017. [https://fileinfo.com/extension/pkl, November 11th, 2019]

[46] STEREOLABS. *Getting Started with ROS and ZED.* [https://www.stereolabs.com/docs/ros/, 4th April, 2020].

[47] MATHWORKS. Rectify stereo images. [https://es.mathworks.com/help/vision/ref/rectifystereoimages.html, 4th April, 2020]

[48] GERIG, G. *Image rectification (stereo).* Spring 2012. [http://www.sci.utah.edu/~gerig/CS6320-S2013/Materials/CS6320-CV-F2012-Rectification.pdf, 4th April, 2020]

[49] GENERALITAT DE CATALUNYA. *Factor de emisión de la energia eléctrica.* 2019. [https://canviclimatic.gencat.cat/es/actua/factors_demissio_associats_a_lenergia/, 4th April, 2020]